REPORT DOCUMENTATION PAGE

Form Approved OMB NO. 0704-0188

1. REPORT DATE (DD-MM-YYYY)	2. REPORT TYPE	3. DATES COVERED (From - To)
01-05-2013	Ph.D. Dissertation	-

4. TITLE AND SUBTITLE
Predicting Attack-prone Components with Source Code Static Analyzers

5a. CONTRACT NUMBER
W911NF-09-1-0239

5b. GRANT NUMBER

5c. PROGRAM ELEMENT NUMBER
611102

6. AUTHORS
Michael Gegick

5d. PROJECT NUMBER

5e. TASK NUMBER

5f. WORK UNIT NUMBER

7. PERFORMING ORGANIZATION NAMES AND ADDRESSES
North Carolina State University
2701 Sullivan Drive
Suite 240
Raleigh, NC 27695 -7514

8. PERFORMING ORGANIZATION REPORT NUMBER

9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS (ES)
U.S. Army Research Office
P.O. Box 12211
Research Triangle Park, NC 27709-2211

10. SPONSOR/MONITOR'S ACRONYM(S)
ARO

11. SPONSOR/MONITOR'S REPORT NUMBER(S)
53488-CS.2

12. DISTRIBUTION AVAILIBILITY STATEMENT
Approved for public release; distribution is unlimited.

13. SUPPLEMENTARY NOTES
The views, opinions and/or findings contained in this report are those of the author(s) and should not contrued as an official Department of the Army position, policy or decision, unless so designated by other documentation.

14. ABSTRACT
No single vulnerability detection technique can identify all vulnerabilities in a software system. However, the vulnerabilities that are identified from a detection technique may be predictive of the residuals. We focus on creating and evaluating statistical models that predict the components that contain the highest risk residual vulnerabilities.

The cost to find and fix faults grows with time in the software life cycle (SLC). A challenge with our statistical

15. SUBJECT TERMS
attack prone, vulnerability prone, static analysis alerts, vulerability

16. SECURITY CLASSIFICATION OF:			17. LIMITATION OF ABSTRACT	15. NUMBER OF PAGES	19a. NAME OF RESPONSIBLE PERSON
a. REPORT	b. ABSTRACT	c. THIS PAGE			Dennis Kekas
UU	UU	UU	UU		19b. TELEPHONE NUMBER 919-515-5297

Report Title

Predicting Attack-prone Components with Source Code Static Analyzers

ABSTRACT

No single vulnerability detection technique can identify all vulnerabilities in a software system. However, the vulnerabilities that are identified from a detection technique may be predictive of the residuals. We focus on creating and evaluating statistical models that predict the components that contain the highest risk residual vulnerabilities.

The cost to find and fix faults grows with time in the software life cycle (SLC). A challenge with our statistical models is to make the predictions available early in the SLC to afford for cost-effective fortifications. Source code static analyzers (SCSA) are available during coding phase and are also capable of detecting code-level vulnerabilities. We use the code-level vulnerabilities identified by these tools to predict the presence of additional coding vulnerabilities and vulnerabilities associated with the design and operation of the software. The goal of this research is to reduce vulnerabilities from escaping into the field by incorporating source code static analysis warnings into statistical models that predict which components are most susceptible to attack.
The independent variable for our statistical model is the count of security-related source SCSA warnings. We also include the following metrics as independent variables in our models to determine if additional metrics are required to increase the accuracy of the model: non-security SCSA warnings, code churn and size, the count of faults found manually during development, and the measure of coupling between components. The dependent variable is the count of vulnerabilities reported by testing and those found in the field.

ABSTRACT

GEGICK, MICHAEL CHARLES. Predicting Attack-prone Components with Source Code Static Analyzers. (Under the direction of Laurie Williams).

No single vulnerability detection technique can identify all vulnerabilities in a software system. However, the vulnerabilities that are identified from a detection technique may be predictive of the residuals. We focus on creating and evaluating statistical models that predict the components that contain the highest risk residual vulnerabilities.

The cost to find and fix faults grows with time in the software life cycle (SLC). A challenge with our statistical models is to make the predictions available early in the SLC to afford for cost-effective fortifications. Source code static analyzers (SCSA) are available during coding phase and are also capable of detecting code-level vulnerabilities. We use the code-level vulnerabilities identified by these tools to predict the presence of additional coding vulnerabilities and vulnerabilities associated with the design and operation of the software. *The goal of this research is to reduce vulnerabilities from escaping into the field by incorporating source code static analysis warnings into statistical models that predict which components are most susceptible to attack.*

The independent variable for our statistical model is the count of security-related source SCSA warnings. We also include the following metrics as independent variables in our models to determine if additional metrics are required to increase the accuracy of the model: non-security SCSA warnings, code churn and size, the count of faults found manually during development, and the measure of coupling between components. The dependent variable is the count of vulnerabilities reported by testing and those found in the field.

We evaluated our model on three commercial telecommunications software systems. Two case studies were performed at an anonymous vendor and the third case study was performed at Cisco Systems. Each system is a different technology and consists of over one million source lines of C/C++ code. The results show positive and statistically significant correlations between the metrics and vulnerability counts. Additionally, the predictive models produce accurate probability rankings that indicate which components are most susceptible to attack. The models are evaluated with receiver operating characteristic curves where each case study showed over 92% of the area was under the curve. We also performed five-fold cross-validation to further demonstrate statistical confidence in the models. Based on these results we contribute the following theory:

Theory: Above a statistically determined threshold, SCSA vulnerability warnings are in the same components as vulnerabilities that are likely to be exploited.

Components that contain security-related warnings identified by SCSA are also likely to contain other exploitable vulnerabilities. Software engineers should systematically inspect and test code for other vulnerabilities when a security-related warning is present. Fortifying these vulnerabilities may facilitate other techniques to identify more undetected vulnerabilities.

Predicting Attack-prone Components with Source Code Static Analyzers

by
Michael Charles Gegick

A dissertation submitted to the Graduate Faculty of
North Carolina State University
in partial fulfillment of the
requirements for the degree of
Doctor of Philosophy

Computer Science

Raleigh, North Carolina

1 May 2009

APPROVED BY:

_____	_____
Jason Osborne	Mladen Vouk
_____	_____
Laurie Williams Committee Chair	Tao Xie

BIOGRAPHY

In his spare time, Michael likes to swim, bike, run, and lift weights.

ACKNOWLEDGEMENTS

Thanks to the altruistic help from the anonymous vendor that afforded me two case studies during my PhD candidacy. They received no credit in my research to protect the anonymity of the industrial partner – so few people would demonstrate such altruism. Also thanks to the Cisco team: Jim, Luis, Mod, Eastern and Western Lisa, Pete, Greg, Omar, Mike, KC, and Wesley. Each contributed greatly to the Cisco case study. Cisco also provided funding for two semesters. Also, thanks to Mladen Vouk and Jason Osborne for spending many hours reviewing my analyses over the past few years. Prasanth and I spent four hours with Malden on a Friday evening. Thanks to Tao Xie for helping me with the new project. His advice is invaluable and I look forward to publishing the new results with him. I spent two summers at Cigital, Inc. where I learned the more about security than I could have anywhere else. A special thanks to Gary, John, Will, Pub, Jeff, Lynn, Matt, and Rob. I also had the opportunity to spend a week at Fortify Software. Brian and Edward were extremely helpful in facilitating my Sendmail scans. Thanks to Laurie for reviewing my work and for the financial support.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

Glossary

Architectural design - (1) The process of defining a collection of hardware and software components and their interfaces to establish the framework for the development of a computer system. (2) The result of the process in (1) [47].

Component – "an entity with discrete structure, such as an assembly or software module, within a system considered at a particular level of analysis" [49].

Error - (1) The difference between a computed, observed, or measured value or condition and the true, specified, or theoretically correct value or condition. For example, a difference of 30 meters between a computed result and the correct result. (2) An incorrect step, process, or data definition. For example, an incorrect instruction in a computer program. (3) An incorrect result. For example, a computed result of 12 when the correct result is 10. (4) A human action that produces an incorrect result. For example, an incorrect action on the part of a programmer or operator [47]. Note: While all four definitions are commonly used, definitions 2, 3, and four can be synonymous with other terms. The second definition of error is synonymous with definition 2 of fault. Definition 3 is synonymous with failure. Definition 4 is synonymous with mistake. *See also* fault.

Execute - to carry out an instruction, process, or computer program. [47]

Execution time - The amount of elapsed time or processor time used in executing a computer program. Note: Processor time is usually less than elapsed time because the processor may be idle (for example, awaiting needed computer resources) or employed on

other tasks during the execution of a program. Syn: run time (3); running time. See also: overhead time. [47]

External metric - External metrics ISO/IEC 9126-2 describes those metrics that represent the external perspective of software quality when the software is in use. The external measures are taken over some predefined period while the software is in use. Values for quantities like time and effort are used as the basis for these measures. These measures apply in both the testing and operation phases. When used during test they are meant to be early predictors of the levels of quality that can be expected once the software is used and operated. These measures generally represent the quality in terms that people use to communicate it [48].

Failure - reliability concept: "The inability of a software system or component to perform its required functions within specified performance requirements" [49].

Failure-prone component - reliability concept: A component that will likely fail due to the execution of faults [85].

We illustrate the difference between faults, failures, vulnerabilities, and attacks in Figure 1.

Fault - "An incorrect step, process, or data definition in a computer program. Note: A fault, if encountered, may cause a failure" [49].

Fault-prone component - "A component that will likely contain faults" [20].

Fault tolerance - (1) The ability of a system or component to continue normal operation despite the presence of hardware or software faults. (2) The number of faults a system or component can withstand before normal operation is impaired. (3) Pertaining to the study of

errors, faults, and failures, and of methods for enabling systems to continue normal operation in the presence of faults.

Internal metric - ISO/IEC 9126 describes those metrics that measure internal attributes of the software related to design and code. These “early” measures are used as indicators to predict what can be expected once the system is in test and operation. Therefore the internal measures are most important to development managers since they are a valuable device for forestalling downstream problems. Internal measures are used to predict the values of corresponding external measures. The internal metrics standard (ISO/IEC 9126-3) shows the relationship between external and internal metrics [48].

Measure - The number or category assigned to an attribute of an entity by making a measurement. *See also* metric. [48].

Measurement scale - A scale that constrains the type of data analysis that can be performed on it. *See also* measure.

NOTE - Examples of scales are: a nominal scale which corresponds to a set of categories; an ordinal scale which corresponds to an ordered set of scale points; an interval scale which corresponds to an ordered scale with equidistant scale points; and a ratio scale which not only has equidistant scale point but also possess an absolute zero [48].

Metric - A measurement scale and the method used for measurement. The word "measure" is used to refer to the result of a measurement. *See also* measure.

NOTES

1. Metrics can be internal or external. *See also* internal metric and internal metric.

2. Metrics include methods for categorizing qualitative data. [48].

Mistake - A human action that produces an incorrect result. *Note:* The fault tolerance discipline distinguishes between the human action (a mistake), its manifestation (a hardware or software fault), the result of the fault (a failure), and the amount by which the result is incorrect (the error) [47].

Natural unit – A unit other than time that is related to the amount of processing preformed by software-based product, such as runs, pages of output, transactions, telephone calls, jobs, semiconductor wafers, queries, or API calls [63].

Operational vulnerability - A vulnerability in the configuration, environment, or general use of the software [23]. For example, an unsafe open()on a file where read/write locks on the same file are possible by other users in the system. System administrators must configure their system so users do not have write permissions to these files. *See also* vulnerability.

Reliability - "The ability of a system or component to perform its required functions under stated conditions for a specified period of time" [49].

Risk - "the potential for realization of unwanted, negative consequences of an event" [81].

Risk analysis -"produces assessments of the loss-probability and loss-magnitude associated with each of the identified risk items, and assessments of compound risks involved in risk-item interactions. Typical techniques include network analysis, decision trees, cost models, and performance models" [12].

Risk assessment –"involves risk identification, risk analysis, and risk prioritization" [12]

Risk management – an emerging discipline whose objectives are to identify, address, and eliminate software risk items before they become either threats to successful software

operation or major sources of software rework" [12]. "The practice of risk management involves two primary steps, risk assessment and risk control, each with three subsidiary steps" [12].

Risk identification - "produces lists of the project-specific risk items likely to compromise a project's satisfactory outcome. Typical risk identification techniques include checklists, decomposition, comparison with experience, and examination of decision drivers" [12]

Risk prioritization - "produces a prioritized ordering of the risk items identified and analyzed. Typical techniques include risk exposure analysis, RRL analysis, and Delphi or group-consensus techniques" [12].

Run - (1) In software engineering, a single, usually continuous, execution of a computer program. See also: run time. (2) To execute a computer program. [47]

Run time - (1) The instant at which a computer program begins to execute. (2) The period of time during which a computer program is executing. (3) See: execution time. [47]

Security - "The protection of system items from accidental or malicious access, use, modification, destruction, or disclosure" [49].

Security design flaw – A vulnerability that occurs from a mistake in the design and precludes the program from operating securely, no matter how perfectly it is implemented by the coders [95]. Examples of design flaws include a lack of or incorrect auditing/logging, ordering and timing faults, and improper authentication [56]. *See also* vulnerability. Security design flaw is shortened to "design flaw" in security jargon.

Security implementation bug – A vulnerability at the code-level of a software system [56]. Bugs are typically easier to fix than design flaws or operational vulnerabilities. *See also*

vulnerability. Security implementation bug is shortened to "implementation" in security jargon.

Software engineering is "the application of science and mathematics by which the capabilities of computer equipment are made useful to man via computer programs, procedures, and associated documentation" [11].

Software life cycle - The period of time that begins when a software product is conceived and ends when the software is no longer available for use. The software life cycle typically includes a concept phase, requirements phase, design phase, implementation phase, test phase, installation and checkout phase, operation and maintenance phase, and, sometimes, retirement phase. *Note:* These phases may overlap or be performed iteratively [47].

Software project – "an organizational effort over a limited period of time, staffed with people and equipped with the other required resources to produce a certain result" [26] .

Software process – "the set of principles, methods, and tools used by a project. A development process defines which activities receive what emphasis, and how and in what sequence they are performed. Accordingly, different process models are used" [26].

Security assurance (or simply assurance) – "confidence that an entity meets its security requirements, based on specific evidence provided by the application of assurance techniques" [8].

Static - Pertaining to an event or process that occurs without computer program execution [47].

Static analysis - The process of evaluating a system or component based on its form, structure, content, or documentation [47]. *See also* static.

Trustworthy – "An entity is trustworthy if there is sufficient credible evidence leading one to believe that the system will meet a set of given requirements. Trust is a measure of trustworthiness, relying on evidence provided" [8].

Vulnerability - An instance of a [fault] in the specification, development, or configuration of software such that its execution can violate an [implicit or explicit] security policy [53].

1. Introduction

Security vulnerabilities can occur because of subtleties and false assumptions in the design or code of a software system [56]. Security engineers often need more time to uncover and fix these problems than their release schedule permits, and so vulnerabilities are inevitably released with the software. Additionally, there are usually fewer security engineers assigned to projects than general reliability engineers, but the security engineers are responsible for testing the security posture of the entire system. Attackers have the advantage of time; they can spend months or years building an exploit for just one area of a software system. Collectively, these conditions necessitate security risk management where security efforts are prioritized to the highest security risk software components to minimize damage to the end user.

Security should be designed and built into the software to prevent attackers from exploiting the software [2]. McGraw [56] uses the term "software security" to refer to the idea of building security into the software. The idea of building secure software implies that good software engineering practices are necessary to properly incorporate security into the software life cycle (SLC) [90]. Techniques that improve software security efforts can alleviate the dependence on protection mechanisms such as firewalls, anti-virus software, and intrusion detection systems as the first and only line of defense. For example, the Morris worm exploited a buffer overflow in the finger service on Unix systems [80]. Software tools such as source code static analyzers by Fortify Software[1], Klocwork[2], and Gimpel[3] analyze

[1] http://www.fortify.com

[2] http://www.klocwork.com

the source code for buffer overflows, among other vulnerabilities, and can be used during the coding phase of the SLC. If a source code analyzer was available to a software engineers that implemented the finger service, they may have identified the buffer overflow and prevented the Morris worm from propagating on the Internet. As Ranum [57] states, security should be built into the software and closely monitored for attacks and thus externally applied protection mechanisms should be used in addition to software security. In this dissertation, we focus on software security. Specifically, we address the previously-mentioned limitations using predictive models to prioritize security efforts to the software components most susceptible to attack.

The cost to find and fix faults grows with time in the SLC [11]. For example, the cost to find and fix vulnerabilities is twice as expensive in late-cycle testing phases (where penetration, system, and function testing is performed) than in the coding phase [11]. Further, the cost to find and fix vulnerabilities is 15 times more expensive in operation than during the coding phase [11]. Early knowledge of which components should receive the most security effort can reduce the number of end-user-installed security patches. The cost for technical support, development of the patch, testing, and deploying a patch is much higher than if security is built into the software at the onset of the software process. Additionally, some users may be unable to deploy a patch to their software systems. For example, a service provider that owns hundreds or thousands of vulnerable software units cannot simply deploy a patch to all of the vulnerable units. The so-called "penetrate-and-

[3] http://www.gimpel.com

patch" technique where end-users and attackers are performing the penetration tests is not an economically viable means to securing software.

No single vulnerability detection technique can identify all vulnerabilities in a software system [96]. For example, performing architectural risk analyses in the design phase of the SLC may not identify insecure permissions for an asset or vulnerable states in protocol implementation code during penetration testing. However, the vulnerabilities that are identified from a detection technique may be predictive of the residuals. We focus on creating and evaluating statistical models that predict the components that contain the highest risk residual vulnerabilities.

A challenge with our statistical models is to make the predictions available early in the SLC to afford for cost-effective fortifications. Source code static analyzers (SCSA) are available during coding phase and are also capable of detecting code-level vulnerabilities. We use the code-level vulnerabilities identified by these tools to predict the presence of additional coding vulnerabilities and vulnerabilities associated with the design and operation of the software. *The goal of this research is to reduce vulnerabilities from escaping into the field by incorporating source code static analysis warnings into statistical models that predict which components are most susceptible to attack.*

The independent variable for our statistical model is the count of security-related SCSA warnings. We also include the following metrics as independent variables in our models to determine if additional metrics are required to increase the accuracy of the model: non-security SCSA warnings, code churn and size, the count of faults found manually during development, and the measure of coupling between components. The dependent variable is

the count of vulnerabilities reported by testing and those found in the field. To meet our research goal, we make the following hypothesis:

Hypothesis: Above a statistically determined threshold, SCSA vulnerability warnings are in the same components as vulnerabilities that are likely to be exploited.

We evaluated our model on three commercial telecommunications software systems. Two case studies were performed at an anonymous vendor and the third case study was performed at Cisco Systems. Each system is a different technology and consists of over one million source lines of C/C++ code. The results show positive and statistically significant correlations between the metrics and vulnerability counts. Additionally, the predictive models produce probability rankings that indicate which components are most susceptible to attack. The models are evaluated with receiver operating characteristic curves where each case study showed over 92% of the area was under the curve. We also performed five-fold cross-validation to further demonstrate statistical confidence in the models. Based on these results we contribute the following theory:

Theory: Components that contain security-related warnings identified by SCSA are also likely to contain other exploitable vulnerabilities. Software engineers should systematically inspect and test code for other vulnerabilities when a security-related warning is present.

In Chapters 2 and 3, we provide background and related work. In Chapter 4 we define vulnerability- and attack-prone components. In Chapter 5 we introduce the static-dynamic coupling effect. In Chapter 6, we provide the Goals, Questions, Metrics, Models, and Responses of our approach. In Chapter 7, we outline our Goal, Questions, Metrics, Models,

and Responses. In Chapter 8, we review our three empirical case studies. In Chapter 9, we provide our research methodology. In Chapter 10, we present our limitations and threats to validity. In Chapter 11, we report the results for three case studies with SCSA metrics. In Chapter 12, we report the results with multiple metrics. In Chapter 13, we show the steps we performed to evaluate our models. In Chapter 14 we provide some discussions about our results. In Chapter 15, we summarize our research contributions, observations, and theory. Finally, in Chapter 16, we discuss our future work.

2. Background

2.1 Defining Vulnerability

Vulnerability - An instance of a [fault] in the specification, development, or configuration of software such that its execution can violate an [implicit or explicit] security policy [53].

Krsul [23] states that a vulnerability is an instance of an error in the specification, development, or configuration of software such that its execution can violate a security policy. Our research is performed in the context of the fault tolerance discipline and thus error is replaced by fault. The usage of fault in the definition of vulnerability simplifies the parallelism of this research to the reliability context. In the IEEE Standard Glossary of Software Engineering Terminology [53], the second definition of error is essentially identical to the second definition of fault (see Chapter 2.1).

Many terms in the security realm are not formally defined by standards bodies such as IEEE. For example, vulnerability, exploit, attack and threat are not defined by the IEEE Standard Glossary of Software Engineering Terminology [53] because it was published in 1990 before the prominence of security in software engineering. This research acknowledges the terms “implementation bugs,” “design flaws,” and “operational vulnerabilities” which are used in practice to describe different categories of vulnerabilities, but are not recognized by international or standards organizations. An objective and repeatable method or taxonomy does not exist to assign vulnerabilities to any of the three categories.

Implementation bugs (implementation vulnerabilities) are security faults that occur in the source code of a software system [95]. Examples of implementation vulnerabilities include buffer overflows, not checking return codes, and not handling unexpected input properly. Source code static analysis tools are created to identify these code-level faults.

A design flaw (design vulnerability) is an instance of a mistake in the design of a software system [42]. Examples of design vulnerabilities include error-handling and recover systems that fail in an insecure fashion, object-sharing systems that mistakenly include transitive trust issues, object-sharing, unprotected data channels, incorrect or missing access control mechanisms, lack of auditing/logging or incorrect logging, and ordering and timing errors. Wysopal et al. further state that design vulnerabilities can occur regardless of how perfectly the code is implemented [95]. Additionally, Dowd et al. [23] state that a design vulnerability occurs when the software is implemented to perform an insecure function.

Operational vulnerabilities are another class of vulnerabilities is suggested by Dowd et al. [23]. Operational vulnerabilities are problems that occur through the operational procedures and general use of software in given environment. For example, an insecure configuration change to a software system in an environment that affords an attacker to exploit the system is an operational vulnerability. Other examples include social engineering attacks, theft, problems with supporting software and computers, and problems caused by automated and manual processes that surround the system.

Security experts view vulnerabilities at different levels of abstraction in vulnerabilities. For example, a buffer overflow could be considered an implementation vulnerability because a developer failed to set a limit on a call to the `strcpy()` C library function in their source

code. The same vulnerability could be considered a design flaw because variables, program state, and function arguments are adjacent on the stack in the architecture of the C programming language and thus apt to be overwritten. The classification of bugs, flaws, and operational vulnerabilities may be helpful in distinguishing which types of vulnerabilities an SCSA can detect.

According to McGraw [56], up to 60% of security vulnerabilities are design flaws. At Microsoft, the infestation of vulnerabilities is a 50/50 split between implementation bugs and design flaws [43]. SCSA analyze code and are therefore more apt at revealing implementation bugs than design flaws and operational vulnerabilities. In security, design and operation problems may require testing or architectural risk analyses for precise identification. The predictive part of this research will attempt to indicate the existence of the more complex and abstract design vulnerabilities and operational vulnerabilities in a component based on the presence of SCSA warnings under the premise that where there are code-level problems there are likely to be other coding, design, and operational vulnerabilities present.

2.2 Calculating Security Risk

The traditional formula for calculating risk in the general reliability realm is probability that the failure will occur multiplied by the impact to the system [11]. In the context of security, the value of risk is similarly calculated by multiplying the threat likelihood by the threat impact [87]. In this research we use a more intuitive definition of risk as proposed in [93]. The risk is calculated as the product of ease of attack multiplied by the value of the asset to an organization and/or an attacker [93].

We will investigate if statistical models can predict which components have the most security risk. The models will rank the vulnerability- and/or attack-prone components by the estimated probability of being vulnerable based on the values that the metrics uses to measure the characteristics of that component. The metrics may indicate that some components are inherently more insecure due to their exposure to un-trusted input. The exposure of a component is best defined by the term attack-surface. According to Howard and Lipner, the attack surface of an application is the union of code, interfaces, services, and protocols available to all users, especially what is accessible by unauthenticated or remote users [44].

2.3 Source Code Static Analyzers (SCSA)

We use source code analysis tool output as one of our candidate metrics for the case study in this paper. A source code analysis tool statically analyzes the content of a software system to detect faults without executing the code [16]. Fault-based testing is the idea of testing for pre-specified faults [60]. Although source code analyzers do not execute software as performed by testing, they do identify specific faults.

We use the term "source code static analyzer" (SCSA) to refer to the use of static analysis tools that analyze source code. Examples of the types of problems identified by SCSA tools include the detection of calls to potentially insecure library functions, bounds-checking errors and scalar type confusion. SCSA tools perform analyses such as semantic, structural, configuration, control- and data-flow analyses. The output of an SCSA tool is a warning. The warnings describe a fault in the software that could lead to a failure. A true positive is a warning that describes a fault that can cause a failure, and a false positive is a warning that

misclassifies code as faulty. Increasingly, SCSA tools are used to identify security vulnerabilities [17].

2.4 Statistical Overview

The input variables and assumptions about the distribution (e.g., Poisson) constitute a statistical model [54]. We also include the data set in which the model is applied to in the definition of a statistical model. Statistical models have been used in many applications. For example, logistic regression can be used to measure thermal distress on the Space Shuttle Challenger O-Rings [54], the Poisson distribution has been used to measure insect counts in insect control experiments [54], discriminant analysis was used by Fisher to classify which species an Iris flower belonged to [83]. These same models have been applied to the software reliability realm (e.g. [52, 77]). The models proposed in this dissertation are fundamentally different than the reliability realm because the dependent variable, security failures, is a distinct subset of the reliability failures that provide deviant operational profiles that afford an attacker to exploit a software system. This size of the data sets for case studies 1 and 2 is 4.2% and 4.7%, respectively, of the entire failure report population. If the populations of vulnerabilities were a large percentage of the non-security faults, then we would then suggest that security prediction models that utilize the same metrics and techniques as those in the reliability realm are too similar for a novel contribution to the field of computer science.

2.4.1 Correlations and Collinearity

A correlation coefficient, r, measures how strongly two variables are linearly related [21]. A correlation coefficient has a value between -1 and 1, inclusive. A weak correlation has a

value between 0 and 0.5 and a strong correlation is greater than or equal to 0.8, otherwise the correlation is moderate [21]. Collinearity is defined as a high degree of correlation between the independent variables of a statistical model [27]. Collinearity occurs when an excessive number of input variables are used to determine an outcome, and the input variables measure the same outcome [27].

2.4.2 Cross-validation and ROC Curves

To evaluate our models, we perform five-fold cross-validation. Five has been shown to be a good value for performing cross-validation [41]. The cross-validation technique validates the R^2 (the fraction of variance explained by the model) by testing the model on data the model has not used before to determine if the model is still effective [94]. The R^2 is called the coefficient of determination in the case of single independent variables – coefficient of multiple determination when multiple independent variables are used. It is rare to have high R^2 values when the dependent variable is categorical [84].

The five-fold cross-validation divides ("folds") the total system components into five groups consisting of approximately equal numbers of randomly-chosen components. One group is used as the test set, and the training set consists of the remaining four groups of components. The model is trained on the training set, and the training analysis is compared to the outcomes of the test set to validate how well the model performs on data that have not been "seen" before. Each of the five groups of components takes one turn as the test set. After the five analyses are performed, the average error is calculated over the five trials. Additionally, we will use receiver operating characteristic (ROC) curves to quantify the goodness of fit of the model to the data set. The y-axis of the ROC curve is the true positive

rate of the predictive model and the x-axis is the false positive rate. An area under the ROC curve equal to 50% indicates that the model has no effective utility to a software engineer. The area should approach 100% to be effective for software engineers.

2.4.3 Classification and Regression Trees (CART)

CART is a statistical technique that recursively partitions data according to X and Y values. The result of partitioning is a tree of groups where the X values of each group best predicts a Y value. The leaves of the tree are determined by the largest likelihood-ratio chi-square statistic. The threshold, or split, between leaves is chosen by maximizing the difference in the responses between any two leaves [83]. For the case study reported in this paper, the X values are the candidate metrics and the Y value is a binary value describing a component as attack-prone (value of one) or not attack-prone (value of zero). Due to the binary dependent variable, the trees we construct are classification trees (as opposed to regression trees). The CART technique has been shown to be useful for distinguishing failure-prone from not failure-prone components in the reliability realm [91].

2.5 The Common Weakness Enumeration

We use Common Weakness Enumeration (CWE) names to identify the vulnerabilities so our analysis can be repeated on other systems with the same vulnerability naming scheme. The CWE [59] is a publicly-available aggregation of security-based vulnerability taxonomies/sources. Examples of the taxonomies the CWE includes are including Seven Pernicious Kingdoms [89], the Preliminary List of Vulnerability Examples for Researchers

(PLOVER)[4], ten from Open Web Application Security Project (OWASP)[5], and the Web Security Threat Classification[6]. The CWE describes software vulnerabilities in a consistently named fashion. For this paper, we use the term "vulnerability" instead of the CWE term "weakness." Each classification of vulnerability contains the classification name and classification ID. For example, with Null Pointer Dereference (476), Null Pointer Dereference is the classification name and (476) is the unique ID given to the classification. The appropriate method of citing a CWE vulnerability is to include both the name and ID.

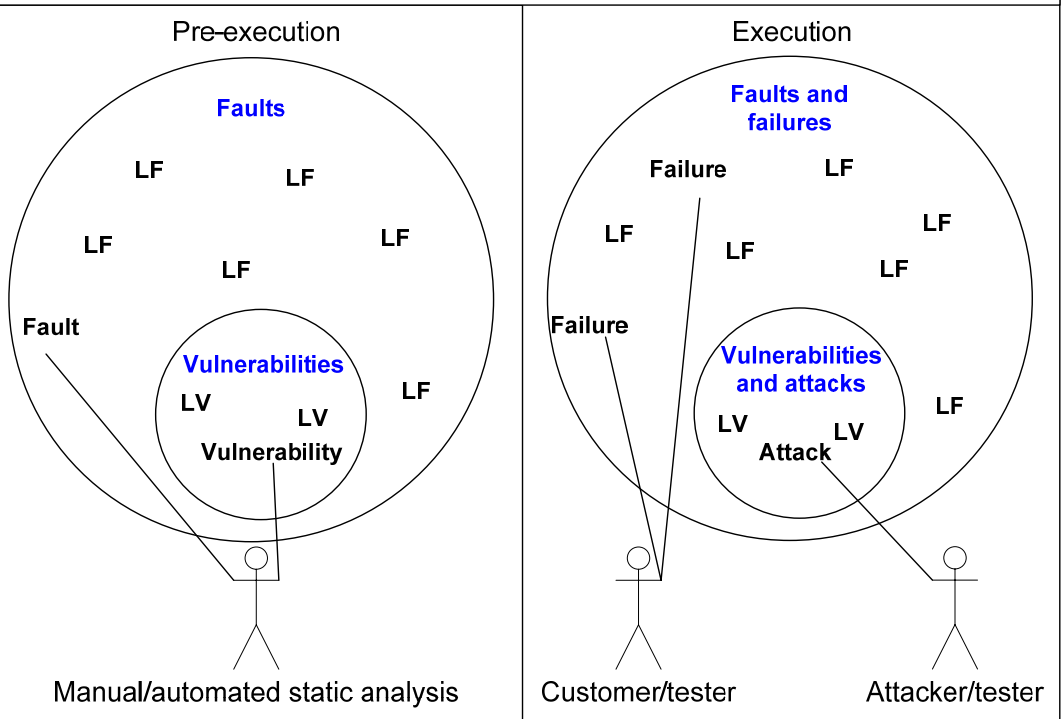
[4] http://cve.mitre.org/docs/plover
[5] http://www.owasp.org/index.php/Main_Page
[6]http://www.webappsec.org/projects/threat/v1/WASC-TC-v1_0.pdf

3. Related Work

Extensive research, including [5, 14, 24, 40, 45, 46, 50, 58, 62, 64-68, 71, 72, 77, 88, 97] has shown that software metrics can be used to identify fault- and failure-prone components and to predict the overall reliability of a system early in the SLC. Recently, a parallel effort has been applied at determining what metrics can predict vulnerable components [30, 74, 78, 86]. Figure 1 shows the difference between faults, failures, vulnerabilities, and attacks. Faults and vulnerabilities refer to problems that are identified statically. Failures and attacks are identified during program execution. Execution occurs during testing and when the system is operating in the field.

Reliability - The ability of a system or component to perform its required functions under stated conditions for a specified period of time [47].

Security - The protection of system items from accidental or malicious access, use, modification, destruction, or disclosure [49].



Fault - *reliability concept*: "An incorrect step, process, or data definition in a computer program. Note: A fault, if encountered, may cause a failure" [47].
Vulnerability - s*ecurity concept*: "An instance of a [fault] in the specification, development, or configuration of software such that its execution can violate an [implicit or explicit] security policy" [53].

Failure - *reliability concept*: "The inability of a software system or component to perform its required functions within specified performance requirements" [47].
Attack - s*ecurity concept*: "The inability of a system or component to perform functions [without violating an implicit or explicit security policy]" [47]. We borrow from the IEEE definition of failure to define attack, but remove the word "required" because attacks can result from functionality that was not stated in the specification.

Figure 1. The difference between reliability, security, faults, failures, vulnerabilities, and attacks. LF=latent fault. LV=latent vulnerability.

3.1 Fault-prone components

Fault-proneness refers the probability that software contains faults [47]. Limited resources preclude software engineers from thoroughly testing a software system and necessitates prioritization of verification and validation (V&V) efforts. To better manage a software system, the system is broken down into smaller parts called components [97]. Software engineers can prioritize efforts at the component level so that quality efforts are focused on the fault-prone components.

Various researchers have different thresholds for the binary classification of components as fault-prone and not fault-prone. Their classification method may be due, in part, to the available metrics and available data set. Nagappan et al. [69] use a conservative normal statistical upper cut-off limit to determine the difference between a fault-prone and a non-fault-prone component. Munson and Khoshgftaar [62] determined the number of faults based on the grouping size for fault-prone and non-fault-prone components. In their research, a fault-prone component has "relatively" more faults than a non-fault-prone component.

One metric used to predict fault-prone components is code churn. Code churn is the count of SLOC that has been added or changed in a component since the previous revision of the software. Nagappan et al. [69] experimented with churn to determine if there was a positive association between churn and defect density. In an analysis with Windows Server 2003, they discovered that there is a correlation and that churn can discriminate between fault-prone and non fault-prone binaries with an accuracy of 89%. They also show that relative churn – churn normalized against lines of code, file count, and file churn – is a better

predictor than the raw value of churn. Elbaum et al. [25] reported that code churn was a more effective indicator or faults than other measures such as the number of people working on the code and the number of change requests to that code.

The size (i.e. the count of source lines of code) of components has also been investigated as a factor associated with the count of faults in seventeen releases of a software system. Ostrand et al. [77] found that the size of a file had the most contribution to a negative binomial model that predicted the count of faults in a software component. When ranking the files in descending order of fault count with size alone, the top 20% of the files on average accounted for 73% of the faults. They also used other metrics to determine if the addition of those metrics increased the accuracy of the model. The other metrics investigated are the file's age, whether or not the file is new during the prior release, number of changes made to the file, and the number of faults in the previous release of the software. The model with the additional metrics identified 83% of the faults over the seventeen releases. Crawford et al. [18] found that size was also the best predictor of all of their complexity metrics for predicting the number of faults in a software system implemented in the C programming language.

3.2 Failure-prone components

Some latent faults require several years to surface and thus information about which components require more testing are revealed after software release [73]. Some research has shown that post-release data can afford software engineers to predict failures. Nagappan et al. [73] have shown that complexity metrics including coupling, McCabe's cyclomatic complexity, count of executable lines of code, fan-in, and fan-out correlate to failures in the

field can successfully predict faults that have surfaced in the field. However, the same combination of metrics they used would not accurately predict failures for all software projects. Binkley et al. [7] studied the number of failures that occurred for four software systems. For each system they correlated the count of failures to a coupling design metric. The coupling metric measures the following three coupling dimensions between modules: referential dependency, structural dependency, and data integrity dependency. They compared the coupling metric to other complexity metrics including cyclomatic complexity, lines of code, count of modules that use a resource, and fan-in, and fan-out. They found that for each system under analysis, the coupling metric had the strongest correlation to the count of failures.

Another metric used to predict failure-prone components is the count of faults identified before the release of the software. Biyani et al. [9] studied the correlation between the number of faults found after release in four releases of an application for high end systems. Their data indicate that modules that were associated with many faults before release are also likely to contain more failures in the field.

3.3 Predicting Fault- and Failure-prone Components with SCSA

Naggappan et al. [97] and Zheng et al. [70] have demonstrated that SCSA warnings can predict fault-prone components. Their work is based on the idea that one technique (e.g., SCSA) alone is insufficient for finding all faults in software [96], but that coding problems reside in the same component as failures identified by testing. Nagappan et al. [97] used the PREfix and PREfast SCSA on Windows Server 2003 code. The dataset consisted of 22 million lines of code and 199 components. SCSA warning density is compared to defect data

which came from testing teams, integration teams, build results, external teams, and third party testers. Customer-reported failures were not included in the analysis. Their results showed they were able to correctly classify 82.91 percent of the components as either fault or non-fault-prone based upon ASA alerts. Type I and Type II errors were not released in the study.

Zheng et al. [97] analyzed three large scale Nortel Network software products. Together the products consisted of more than three million lines of code. They compared warning density to results from over 200 inspectors and testers and from customer-reported failures. The primary SCSA used in their study was FlexeLint[7].

Zheng et al. [97] analyzed their work in the context of the Orthogonal Defect Classification [97]. Faults can be categorized based on when they are introduced in the SLC. IBM's Orthogonal Defect Classification (ODC) [97] has eight different types of faults. They are: Algorithm, Documentation, Checking, Assignment, Function, Interface, Build/Package/Merge, and Timing/Serialization. Zheng et al. correctly classify 83.3 percent of the components as either fault-prone or non-fault-prone. They furthered their study to for their model to include SCSA warnings and the number of test failures to correctly identify 87.5 percent of the components. Thirty-three percent of the fault-prone components were incorrectly classified as non-fault-prone components. Additionally, they used ASA alerts and normalized test failure density to correctly classify 91.7% of the modules, but there were 22 percent false negatives among the fault-prone components. Zheng et al. [97] conclude that SCSA are good for predicting Assignment and Checking errors, but for their data the

[7] http://www.gimpel.com/html/products.htm.

other six ODC types were not identified by SCSA. Their data indicated that SCSA warnings are good indicators of fault-prone components in the general case (beyond simply Assignment and Checking errors).

3.4 Vulnerability Prediction Models

Neuhaus et al. [74] have investigated statistical models that predict vulnerable components. They created a software tool, Vulture, that mines a bug database for data including libraries and APIs to identify vulnerable components. They performed an analysis with Vulture on Bugzilla, the bug database for the Mozilla browser, using imports and function calls as predictors. In their research, they predict the presence of a vulnerability based on the specific *domain* of the component. In their setting, the most vulnerable domain was domain in which Mozilla performed or was capable of scripting-related operations. They identified 45% of the vulnerable components in Mozilla with a 30% false positive rate.

Ozment and Schechter [78] investigate if most of the vulnerabilities in OpenBSD reside in legacy code or in the new code. Approximately 62% of the vulnerabilities are in the legacy code of OpenBSD. Ozment and Schechter [78] also consider code churn as a predictor of which files are most likely to be vulnerable. Their results indicate that there is no significant correlation between code churn and vulnerability count. Their correlations are performed at the version level of OpenBSD, while our studies and those of Neuhaus et al. [74] are performed at the component-level. The version-level of a software system may too high-level to make statistical correlations; analyses at the component-level may offer more meaningful results.

3.5 The Effect of Complexity on SCSA

Walden et al. [92] analyzed the effect of code complexity on the Fortify Software's SCSA, Source Code Analyzer (SCA) 4.5.0, to detect format string vulnerabilities. The complexity metrics they analyzed are the count of source lines of code (SLOC) and McCabe's cyclomatic complexity [55]. SCA identified 22 of 35 (63%) format string vulnerabilities in the open source Linux software systems tested. They noted that the detection of format string vulnerabilities decreases as the complexity of the source code increases, but found that complexity was not the cause of the vulnerabilities not detected by SCA.

Walden et al. identified two causes for the 37% false negative rate. First, four of the thirteen (31%) format string vulnerabilities that were not detected were not implemented in SCA's rule set. However, Walden et al. claim that developers could implement their own rule and add it to SCA's rule set to identify the format string vulnerabilities. Nine of the thirteen (69%) of the format string vulnerabilities were not detected by SCA because of a fault that existed in the SCA. The fault was reported to Fortify Software.

If developers can implement good rules for format string vulnerabilities and they use the fixed version of SCA, then the format string vulnerabilities may not have escaped into the field. The limitation of this work is that the vulnerabilities studied are only format string vulnerabilities and not to all vulnerabilities in the SCA rule set. One of the metrics in our research is coupling. Coupling is used to predict if vulnerabilities not identified by SCSA are in the same components as SCSA warnings. As mentioned in Chapter 3.2, complexity is

correlated to field failures and although SCSA may detect the coding vulnerabilities their predictive power may be enhanced in the most complex components.

4. Vulnerability- and Attack-prone Components

While Krsul [53] defined vulnerability in accordance to the fault tolerance discipline, we define attack for this dissertation. We define attack as a subset of failures in the same way Krsul [53] defined vulnerability as a subset of faults. The term attack is therefore represents a *security failure*. Likewise a vulnerability can be referred to as a *security fault*. Additionally, we define the terms vulnerability- and attack-prone components in this dissertation as first proposed in [30]. The definitions are now stated:

Attack - The inability of a system or component to perform functions without violating an implicit or explicit security policy [36]. We borrow from the ISO/IEC 24765 [49] definition of failure to define attack, but remove the word "required" because attacks can result from functionality that was not stated in the specification.

Attack-prone component - A component that will likely be exploited [30].

Vulnerability-prone component - A component that is likely to contain one or more vulnerabilities that may or may not be exploitable [30].

Fault-prone prediction models can estimate which components are fault-prone, but if the faulty code is never executed it will not be prone to failure. The inspection of all fault-prone components may cause the development team to expend valuable and limited verification resources on low risk areas of the code that may be of high quality and/or may rarely or never be used by a customer. Failure-prone prediction models, based on a customer's operational profile, and historical failures from the field, can further guide fault-finding efforts toward low quality components that are likely to cause problems for the end user.

Security can be viewed as a subset of reliability [90] and thus the reliability-type models may be applicable in the security context. One contribution of this dissertation is definitions of vulnerability- and attack-prone components. A **vulnerability-prone component** is analogous to a fault-prone component in that vulnerabilities may remain latent (similar to faults) until encountered by an attacker (or tester/customer) in an executing system. The vulnerabilities in a vulnerability-prone component can include a wide range of severity and likelihood of exploitation. An example of a vulnerability-prone component is a component that contains many vulnerabilities identified by static inspections. The static inspections include peer reviews and analyses from source code and binary analyzers that analyze the software without executing the software. While these vulnerabilities can have any severity and likelihood of being exploited, we reserve the term attack-prone component for when a vulnerability is actually exploited during execution. Execution occurs during testing and when the software is running in the field.

A similar relationship between vulnerability-prone and attack-prone components exists as with fault- and failure-prone components. An **attack-prone component** is a vulnerability-prone component if an attacker will likely exploit one or more vulnerabilities in that component. Vulnerability-finding techniques can cause security experts to expend valuable and limited security resources on low risk areas of the code that may be adequately fortified, may be uninteresting to an attacker, or contain difficult-to-exploit vulnerabilities. Attack-prone prediction models based on test failure data and attacks in the field can make vulnerability-finding efforts more efficient and effective by identifying those components with the highest security risk. In the context of testing, if a tester discovers a buffer overflow

during runtime, we say they have attacked the system. Although the tester may not have exploited the buffer overflow to cause a denial-of-service or to inject code that escalates their privileges, the failure is a proof of concept that the system can be attacked. While failures in the reliability realm are dependent on the operational profile, attacks can occur anywhere in a software system regardless of the operational profile.

5. The Coupling Effect

5.1 Background

The coupling effect states that "test data that distinguishes all programs differing from a correct one by only simple errors is so sensitive that it also implicitly distinguishes more complex errors." [19]. Evidence of the coupling effect was first shown in the context of mutation testing where simple test sets could identify both simple and complex faults [19]. The coupling effect indicates that when a software engineer identifies a simple fault, then they should also perform a systematic search for a more complex fault or failure [19]. Therefore, DeMillo et al. [19] conclude that simple errors compound in more massive error conditions [19].

Offutt [75] observed a derivative of the coupling effect called the mutation coupling effect with a mutation testing experiment on three programs: MID, TRITYP, and FIND. In mutation testing, a simple fault can be created by a mutant operator changing a normal program. A complex fault is mutant program with two or more mutations. A mutant with two mutations is a 2-order mutant. The experimental hypothesis was that a test-data set that is mutant-adequate for all 1-order mutants is also mutant-adequate for 2-rorder mutants. Offutt found that the test data developed to kill 1-order mutants was also successful at killing 2-order mutants. For the MID program, 19,110 2-order mutants were created; 18,961 of them were killed. Of the remaining mutants, 145 of them were equivalent to the original program and four were alive. The 2-order mutants that remained alive were not dissimilar to 1-order mutants that could be killed with typical data sets. He also found that the same set of

test data developed for 1-order mutant actually killed a higher percentage of mutants when applied to 2-order mutants.

Offutt [76] contributed the idea of the *mutation coupling effect* to provide evidence of the coupling effect in the context of mutation testing as opposed to stating that the coupling effect applies to all faults, all programs, and all test data sets. The mutation coupling effect states that complex mutants are coupled to simple mutants in such a way that a test data set that detects all simple mutants will detect a large percentage of complex mutants [76]. Offutt [76] defines a simple mutant as a program that expresses only one mutation and a complex mutant is a mutant program that consists of multiple mutations. The use of "detect" in the mutation coupling effect definition indicates that a test case that "kills" a simple mutant will also cause the complex fault to be realized. Offutt describes the coupling of simple and complex faults as being in the same path of execution in the following idea:

> *Of course, the fact that two mutants are executed on the same path does not necessarily mean that they interact in any meaningful way, but such interactions are difficult to determine analytically [76].*

The relationship between the simple and complex faults does not include the proximity of the faults in terms of their physical location in a software component. In our research, we investigate the component dimension between simple and more complex faults as described in the next chapter.

5.2 The Static-dynamic Coupling Effect

Young and Taylor [96] state the idea that "no single technique is capable of addressing all fault-detection concerns." More recently this idea has been applied to the security realm. According to McGraw [56] "No individual touchpoint or tool can solve all of your software security problems. [56]". A touchpoint, in this context, is a software security best practice applied to a software artifact [56]. McGraw [56] also adds that "static analysis tools can find bugs in the nitty-gritty details, but they can't even begin to critique design". McGraw exemplifies his claim in a scenario where a software engineer is developing a funds transfer application. Although the SCSAs can find some faults in the software, he claims the tool cannot abstract the design of the system based on the code to warn the user to strengthen user password requirements [56]. While SCSA can measure coupling between components which represent the low-level design, they currently do not interpret high-level design issues. Furthermore, Chess and West state that "for a static analysis tool to catch a defect, the defect must be visible in the code. This might seem like an obvious point, but it is important to understand that architectural risk analysis is a necessary complement to static analysis." Although SCSA analysis are useful, software engineers cannot exclusively rely on their ability to identify all types of software vulnerabilities.

Static analysis is the process of evaluating a system or component based on its form, structure, content, or documentation [47] Static analyses include binary code analyses and peer review code inspections. As mentioned, we investigate if SCSA faults and more complex vulnerabilities identified during execution are in the same component and so we further refine this hypothesis to be specific to SCSA.

Our research investigates if there is an association, in the context of components, between the count of SCSA warnings, a static (internal) metric, and the count of failures identified during execution, a dynamic (external) metric. This research leads us to the following hypothesis:

Hypothesis: Above a statistically determined threshold, SCSA vulnerability warnings are in the same components as vulnerabilities that are likely to be exploited. We call this the static-dynamic coupling effect.

We avoid calling the association between faults identified during static and dynamic analyses as simply the coupling effect because the coupling effect is defined in the context of program execution. However, DeMillo et al. [19] clarify their definition of the coupling effect in the idea that "In other words, complex errors are coupled to simple errors. [19]"

The coupling effect does not indicate that simple and complex faults are located in the same component. As mentioned, the simple and complex problems are identified on the same path of execution during mutation testing which does not necessarily require that the component in which they reside is a factor incorporated into the idea of coupling. However, if we assume that the location of faults is a factor in the coupling effect, then the static-dynamic coupling effect provides some support for the more general coupling effect. In this dissertation we investigate evidence of the SCSA fault-dynamic coupling effect which provides evidence of the more general static-dynamic coupling effect where faults found by manual or binary static analyses are in the same component as failures identified by testing.

The hypothesis is more of a statistical indication than an absolute truth. However, if the statistical indication is plausible, then we may improve the security assurance of a software

system by systematically looking for other types of faults that are not identifiable by SCSA in the same component as the SCSA-identified faults. In this dissertation we do not distinguish between simple and complex faults as done in research efforts for the coupling effect (e.g. [19, 75, 76]). Instead, we distinguish faults by the method in which the faults are identified – either statically or dynamically.

6. The Goal, Question, Metric Approach

When you can measure what you are speaking about, and express it in numbers, you know something about it; but when you cannot measure it, when you cannot express it in numbers, your knowledge is of a meager and unsatisfactory kind: it may be the beginning of knowledge, but you have scarcely, in your thoughts, advanced to the stage of science.

— Lord Kelvin

Measurement can indicate progress toward a goal, provide a means to substantiate corrective action based on an assessment, and evaluate the impacts of actions [4]. Purposeful measurements require that a goal set out by a software organization be traced to the organization's data and then be interpretable by the stakeholders of that organization [4]. The goal of the software organization should be clear as to determine what information is needed to achieve the goal. The information should be quantifiable and when analyzed should indicate if the goal has been achieved. The Goal Question Metric (GQM) approach [4] defines a top-down framework for defining a goal, stating questions about the goal, and then declaring what metrics are used for measuring the goal. The result of the application of the GQM approach is a specification that outlines how measurement should be performed and how to interpret the measurements in such a way that the measurements can be traced back to the goal.

The goal is proposed at a conceptual level of a software organization. Goals can be made for objects that include products, processes, and resources. A goal should specify a purpose,

the object that will be measured, a viewpoint, and an issue that is of interest (e.g., a security issue). An example is "Improve the timeliness of vulnerability identification from the security team's viewpoint." In this example, the purpose is to improve, the issue is the timeliness, the object to be measured is vulnerability identification, and the viewpoint is from the security team.

The goal is refined by multiple questions that are defined at the operational level of a software organization. The questions should characterize how the achievement of the goal will be performed. Questions should also characterize the object set forth by the goal to indicate a potential problematic issue with the object. From the example of the goal previously stated, a question could be "How many vulnerabilities are identified before release?" Multiple questions can be associated with a single goal. Lastly, the answers the questions indicate if the goal has been achieved.

The questions are answered with metrics. Metrics are associated with a data set that allow for a quantitative answer to the question. Basili et al. [4] do not indicate that a qualitative analysis is possible, but we do not limit the usage of GQM to quantitative analyses. Multiple metrics can be associated one question. Also, one metric can be used to answer multiple questions. A possible metric in our example is the count of vulnerabilities identified before and after the release of the software. Another metric could be the count of high risk vulnerabilities that are identified in the field.

Basili et al. [6] have since refined the GQM approach to include the concept of a model and a Response. The model is used to interpret the values of the measurements performed with the metrics. For example, if model indicates that a component is more likely to be

attacked if there are more than five vulnerabilities in it, then the component should receive more security attention. If there are less than five vulnerabilities, then security efforts should not be prioritized to those components. In this case five was arbitrarily chosen, but could be calculated by risk management decisions.

The Response is a suggestion on what action should be carried out by software engineers based on the result of the model. For example, the response could include an initial analysis of why vulnerabilities are injected into the software. A check to determine if software engineers are following corporate security requirements and policies could be initiated by management if components are many vulnerabilities. An investigation of how much cost will be associated with the corrective actions could be applied in the response.

The GQM approach prevents software engineers from a bottom-up approach where the metrics drive the goals. A bottom-up approach may be cumbersome in that there are many metrics to observe and knowing which metrics are the most important is challenging without first having a conceptual goal that provides a linkage between the metrics and the goal. We refer to the Goal, Question, Metric, Model, Response framework as GQM-MR in this dissertation.

7. Research Goals, Questions, Metrics, Models, and Responses

The research approach for this dissertation is now discussed in accordance to the GQM-MR-C framework. By referring to post-development in this dissertation, we refer to all phases in the SLC that occur after the coding phase. These phases are referred to as "Development test", "Acceptance test", and "Operation" according to Boehm [11]. These phases include penetration, system, and function testing practices. Also included in post-development are failures that occur in the field reported by customers and third-party researchers. All metrics discussed in our research are on a per component basis. The metrics are numbered for each question. A metric that is used for multiple questions has the same number.

The phase in the SLC at which software engineers assess the security risks is a critical factor for improving the security assurance of a software system. The cost to find and fix vulnerabilities grows with time during the SLC [11]. If a software organization finds and fixes all vulnerabilities late in the SLC, they may not be able to afford the high costs to fix a vulnerability. Further, finding a requirements or design vulnerability post-development may require that substantial and difficult modifications be required to fix the vulnerability. If the vulnerability necessitates a large modification of the system, then software engineers may not find an effective security solution or the solution may require so many changes that all changes are not secured. In such a scenario, a software organization may elect to release the vulnerable software and address the vulnerabilities before the next release of that software system. Therefore, early security efforts are preferable as more time for is available for

vulnerability identification, assessment, and mitigation strategies to yield a secure software system.

Goal. *To reduce vulnerabilities that are most likely to be exploited from escaping into the field by using predictive models that utilize metrics available early in the SLC.*

There are two dimensions in our goal: prevention and prediction. The statistical models predict which components are most susceptible to attack. Software engineers can prioritize vulnerability identification and fortification efforts to attack-prone components to mitigate the chances of a successful attack on a software system.

We must first determine in which phase(s) of the SLC the software organizations we partner with identify vulnerabilities. As mentioned, our objective is to identify vulnerabilities during the coding phase. The extreme cases of where the software organization can find vulnerabilities are now mentioned to illuminate the value of our research goal. If the software organizations find all vulnerabilities during the requirement phase, then the predictive models will be of no value. If all vulnerabilities are identified in the field, then the predictive models will save the software organization the costs from finding and fixing vulnerabilities and any customer-related costs. A more likely scenario would be that the software organizations identify vulnerabilities in each phase of the SLC. In this case, the predictive model would benefit the software organization because vulnerabilities are identified post-development.

Question 1. What is the ratio of vulnerabilities identified during the coding phase to those identified post-development?

Metric 1. Count of vulnerabilities identified in the requirements, design, and coding phases.

Metric 2. Count of vulnerabilities identified during testing and those reported in the field.

Our second question is used to answer the hypothesis of this dissertation stated in Chapter 1. If SCSA warnings are in the same components as vulnerabilities not identifiable by SCSA, then software engineers should simultaneously identify and fortify other types of vulnerabilities while addressing SCSA faults. A Spearman rank correlation between SCSA warnings and vulnerabilities identified during post-development will provide the strength of the association between the different types of vulnerabilities. A positive and significant correlation between SCSA warnings and vulnerabilities identified during operation will give support to the SCSA fault-dynamic coupling effect. The use of SCSA satisfies our goal's requirement that the predictions be performed early in the SLC. Also SCSA offer an objective and repeatable means to identify vulnerabilities in source code. Therefore, the results of our analysis can be repeated with the less subjectivity than if performed by manual effort.

Question 2. Are SCSA warnings in the same components as vulnerabilities identified during system execution (i.e., post-development)?

Metric 1. Count of vulnerabilities identified in the requirements, design, and coding phases.

Metric 2. Count of vulnerabilities identified during post-development.

Metric 3. Count of SCSA warnings.

We now focus on the predictive aspect of the statistical models. The metrics used in our Goal that have significant correlations to vulnerability counts identified post-development phases of the SLC will be candidate metrics in the predictive model.

Limited resources (e.g. budget, person hours, and time) preclude software engineers from finding and fixing all vulnerabilities in a software system. This limitation necessitates risk management where risks are ranked in descending order in an effort to prioritize security efforts to the vulnerabilities that cause the most business impact. We create and evaluate statistical models that predict which components are attack-prone to prioritize security efforts to the attack-prone components.

For Model 1, if attack-prone components are also components that have an SCSA warning count that is greater than a threshold defined by the model, then the predictive power of SCSA warnings also supports the SCSA fault-dynamic coupling effect. If the attack-prone components are below the threshold, then there is no support for the SCSA fault-dynamic coupling effect. Software engineers can use to the model to guide prioritization of vulnerability identification and fortification efforts to the components that are most susceptible to attack. The strategy of creating a model with thresholds and taking action based on the thresholds has also been performed in the reliability realm (e.g. [91]).

Model 1. Predictive model that distinguishes between attack-prone and not attack-prone components based on Metric 3.

if | count of SCSA warnings > w |

where w is a threshold count of SCSA warnings in a component that indicates the component is attack-prone. The statistical model will determine the value of w.

then component is attack-prone

if | count of SCSA warnings < w |

where w is a threshold count of SCSA warnings in a component that indicates the component is not attack-prone.

then component is not attack-prone

Prior research efforts in the reliability realm (e.g. [45, 52, 62, 77]) have shown code churn and size and complexity can predict fault- and failure-prone components. However, these models that use these metrics have not shown that a single metric can identify all a software system's fault- and/or failure-prone components. Further, there is no evidence that 100% fault- and/or failure-prone component identification be can achieved without misclassifications. The misclassifications from the models occur when the models incorrectly classify fault-prone components as not fault-prone and not fault-prone components are misclassified as fault-prone. We therefore investigate if code churn and size, coupling, and faults found manually are associated – as measured by the Spearman rank correlation -- with vulnerabilities identified post-development to improve the identification of attack-prone components. If we assume that security is a subset of reliability, then these metrics may also identify vulnerability- and attack-prone components. We add Metrics 4-7 in Model 2 to determine if these candidate metrics increase the accuracy of Model 1.

Question 3. Can the following additional metrics help the predictive models: code churn and size, complexity, and faults found by manual?

Metric 1. Count of vulnerabilities identified during requirements, design, and coding.

Metric 2. Count of vulnerabilities identified during post-development.

Metric 3. Count of SCSA warnings.

Metric 4. Count of added and changed SLOC[8].

Metric 5. Size (in terms of SLOC).

Metric 6. Count of faults identified during manual inspections.

Metric 7. Coupling as measured by how much one component uses another component.

Based on Question 3, we create a model, Model 2, to determine if the additional candidate metrics can improve the accuracy of Model 1.

Model 2. Predictive model that distinguishes between attack-prone and not attack-prone components based on Metrics 1-7.

if | metric values > w, x, y, z |

where w, x, y, z are the thresholds of the metrics in that component that indicate the component is attack-prone. The statistical model will determine the values of w, x, y, and z.

then component is attack-prone

if | metric value < w, x, y, z |

where w, x, y, and z are values of the metrics in that component that indicate the component is not attack-prone.

then component is not attack-prone

Response for Models 1 and 2. Prioritize security identification and fortification efforts to the attack-prone components. Determine the reason why the exploitable vulnerability or vulnerabilities are injected into the software.

A cost analysis would show how much time should be spent analyzing components that are classified as attack-prone by the model. The probability ranking for the security effort

[8] SLOC is measured by the number of executable lines of source code.

prioritization should include only the components that are most susceptible to attack. A component that is not attack-prone, but is flagged by the model as attack-prone is a false positive. If false positives are assigned a high probability in the probability ranking, software engineers may become reluctant in adopting the model due to wasted effort on components with low security risk.

The null (H_{01}) and alternative (H_{A1}) hypotheses are now stated to answer Question 2. Two conditions must be met for us to not reject the alternative hypothesis. First, the correlations between the metrics and vulnerabilities counts must be positive and significant. Second, SCSA warnings make statistically significant contributions to the predictive models. The goal, questions, metrics, models, response, and cost analysis are summarized in Table 1.

H_{01}: *Above a statistically determined threshold, SCSA vulnerability warnings are not in the same components as vulnerabilities that are likely to be exploited*.

H_{A1}: *Above a statistically determined threshold, SCSA vulnerability warnings are in the same components as vulnerabilities that are likely to be exploited.*

Table 1. The GQM-MR approach summarized for this research.

Goal	To prevent vulnerabilities that are most likely to be exploited from escaping into the field by using predictive models that utilize metrics available early in the SLC
Question 1	What is the ratio of vulnerabilities identified during the coding phase and those identified post-development?
Metric 1	Count of vulnerabilities identified in the requirements, design, and coding phases.
Metric 2	Count of vulnerabilities identified during post-development.
Question 2	Are SCSA warnings in the same components as vulnerabilities identified during system execution (i.e., post-development)?
Metric 1	Count of vulnerabilities identified in the requirements, design, and coding phases.
Metric 2	Count of vulnerabilities identified during post-development.
Metric 3	Count of SCSA warnings.
Model 1	Predictive model that distinguishes between attack-prone and not attack-prone components based on Metric 3.
Question 3	Can the following additional metrics help the predictive models: code churn and size, complexity, and faults found by manual?
Metric 1	Count of vulnerabilities identified in the requirements, design, and coding phases.
Metric 2	Count of vulnerabilities identified during post-development.
Metric 3	Count of SCSA warnings.
Metric 4	Count of added and changed SLOC
Metric 5	Size (in terms of SLOC)
Metric 6	Count of faults identified during manual inspections
Metric 7	Coupling as measured by how much one component uses another component
Model 2	Predictive model that distinguishes between attack-prone and not attack-prone components based on Metrics 1-7
Model 1 and Model 2 Response	Prioritize security identification and fortification efforts to the attack-prone components. Determine the reason why the exploitable vulnerability or vulnerabilities are injected into the software.

8. Three Industrial Case Studies

We conducted case studies of three large commercial telecommunications systems. The first two case studies were conducted with an anonymous vendor. The case studies for this vendor are referred to as Case Study 1 and Case Study 2 for the remainder of this dissertation. The third case study is an analysis on a Cisco software system and is referred to as Case Study 3 in this dissertation. Details of the Cisco system are not provided for confidentiality reasons.

Good metrics are those where measurements of the software are objective, reproducible, and empirical [48]. For example, a software engineer can use a tool to objectively obtain the quantity of source lines of code (SLOC) in a file. The measurements that provide the metrics should also be performed easily and economically [48]. Additionally in our case studies, we required the metrics to be accessible early in the SLC to afford software engineers a means to prioritize security efforts during development. Although SCSA is the primary metric for our analysis, we hypothesize that other metrics could be beneficial for predicting attack-prone components and thus investigate these metrics in Chapter 12. The three case studies will be discussed in the following three subsections.

8.1 Case Study 1

We analyzed data from a large commercial telecommunications software system that had been deployed to the field for two years [35]. The system contained 38 well-defined components whereby each component consisted of multiple source files. A full set of information necessary for our analysis was only available for 25 (66%) of the components of the system, and thus the study focuses on those components. The 25 components we

analyzed summed to approximately 1.2 million lines of code. All faults in the failure reports have since been fixed.

The metrics used in our analysis include failure reports, SCSA warnings, and the count of churn (the sum of added and changed lines of source code) and SLOC per component. The failure reports included pre- and post-release failures. A pre-release failure for our study is a failure discovered by an internal tester during feature and/or system robustness testing. A post-release failure indicates that a failure occurred in the field and was reported by a customer. Both the pre- and post-release failure reports explicitly identified the component where the solution was applied. Information in the failure reports gave details on log output, how to reproduce the failure, stack traces when applicable, severity, impact to end users, test output, and brief general strategies on how to remedy the problem.

An SCSA analysis was performed on the system by FlexeLint[9]. Although FlexeLint is a reliability-based SCSA, we sought to determine if the full set of defect types identified by the tool could be warnings of security vulnerabilities on a per-component basis. Additionally, we classified some of the FlexeLint defect types as security-related (see Section 8.1.1) and used both the security-related warnings and the reliability-related warnings for predictors in our models. For our analysis we were provided with both audited and un-audited output from FlexeLint. The audited output, compiled by the software organization's outsourced manual auditing service, contained an enumeration of true positive warnings. For each warning, the report would contain a brief general description about the warning, a mapping to the component containing the fault, and the impact/severity as defined by the auditors, and a

[9] http://gimpel.com/

code fragment. The un-audited report included all true and false positive warnings and the file in which the warning was found. A file path was given in the un-audited report that was used to map to the component. The pre- and post-release failure reports and the un-audited/audited FlexeLint output provided the numbers of failures and warnings for each component which was sufficient for our analyses. There were 55,024 warnings produced by FlexeLint. The outsourced auditing service classified 302 (0.55%) warnings as true positives. We did not have enough information to determine which of the warnings had identified the faults eventually found by the pre- and post-release failures. Churn and SLOC were reported by the software organization for each of the components we analyzed.

8.1.1 Classification of SCSA Warnings According to the Common Weakness Enumeration

We created three groups of FlexeLint warnings (see Table 2) that could serve as indicators of security problems: buffer overrun, memory leak, and null pointer. The following three SCSA warning groups are considered security-oriented because security engineers also considered the same types of problems identified by testing as security-related: buffer overruns, memory leaks, and null pointers. We provide the FlexeLint warning types, identified by their unique codes in Table 2. The warning descriptions given at the FlexeLint website[10] and the descriptions given in the audited FlexeLint report provided enough information on which warning type belonged to the warning group and CWE classification. The three warning groups together represent 16 of the 2000 FlexeLint warning types.

[10] http://www.gimpel-online.com/MsgRef.html

Table 2. Warning groups with CWE name and warning codes.

Warning group	CWE classification (ID)	FlexeLint warning codes
buffer overrun	Buffer Errors (119) Stack Overflow (121)	415, 416, 419, 420, 661, 662, 669, 670
memory leak	Resource Exhaustion (400)	423, 429, 672, 1540
null pointer	Null Pointer Dereference (476)	412, 418, 613, 668

Approximately 72.4% of the audited warnings are categorized into the three security-based warning groups. When analyzing the un-audited warnings, only 23.5% of the warnings were either buffer overflows, memory leaks, or null pointers. When referring to the combination of three security groups of security warnings in our analyses, we will use the term "total security warnings." Percentages of the security-based warnings as compared to the overall number of warnings are given in Table 3. Buffer overflow warnings constitute approximately half of all of the audited warnings. However, when considering the un-audited warnings, only 1.7% of the warnings are related to buffer overflows.

Table 3. Audited and un-audited SCSA warnings from FlexeLint.

Audited and Un-audited	SCSA security-based warnings	% Total warnings
Audited	buffer overrun	50.1%
	memory leak	14.1%
	null pointer	8.2%
Un-audited	buffer overrun	1.7%
	memory leak	2.4%
	null pointer	19.4%

8.1.2 Failure Report Classification

The author and additional research student, doctoral students in software security, independently reviewed each of the 1255 pre- and post-release failure report to determine which failure reports were security problems. Based on the failure reports we developed criteria that identified which failures were non-security problems and which were indicative of security problems. Some failure reports were explicitly labeled as security problems (approximately 0.5%) by either internal testers or security engineers. We analyzed all other failure reports that were not explicitly labeled a security problem. We found that many reports contained the following keywords that are often seen in security literature: crash, denial-of-service, access level, sizing issues, resource consumption, and data loss. These keywords increased our suspicion of whether or not the failure could be a security problem, even though it was not previously labeled as a security problem.

We compiled a list of these keywords and used it to match against all failure reports that were not explicitly labeled as a security problem. We excluded from our analysis any failure report that indicated that the problem was not reproducible (6.4%) or did not contain enough information about the failure to adequately understand the fault or declare it as a security problem (0.6%). The criteria for a failure report to be classified as a security problem are now listed:

- *Remote attacks*. The failure reports explicitly indicated when the failure was due to a remote user or machine. Pre- and post-release failure reports that contained the security keywords and could be remotely initiated had the highest confidence of indicating an exploitable vulnerability.

- *Insider attacks*. If the failure report did not indicate that the failure was due to an external user or machine, then we looked for attacks that did not require remote access to the system. For example, one report indicated that an insider attack was possible if a disgruntled employee was to abuse a privilege in the system.
- *Audit capability*. Weak or absent logging for important components was considered a security vulnerability. An example of an important log that was not working properly involved loss of a financial transaction that may result in an attacker obtaining a service for free. The absence of logs has been demonstrated as a security problem when audits are required to identify an attack [79].
- *Security documentation*. We also considered if the fundamental principles of software security were followed. For instance, in two failure reports, the testers indicated that the problem would occur if the users were not "well-behaved," which breaks the principle of Reluctance to Trust [3]. Additionally, we also looked at documented vulnerabilities descriptions (e.g. those listed in the Common Weakness Enumeration - see Chapter 2.5) could apply, or if any documented attack patterns [42] could match to the software.

After filtering for security vulnerabilities, the author and additional doctoral student compared their findings, settled differences, and then reported the final results to the software organization's security engineer. The security engineer audited our report and eliminated false positives (6.8%) from our report. False positives were reliability faults that we claimed to be security vulnerabilities. The number of vulnerabilities in our analysis 46 (3.7%) of the total failure reports were classified as security failures. Any of the failure reports that we

misclassified as non-security problems are false negatives in the study. We used the failure reports that were verified as security problems in our statistical analyses. A security-based failure represents the presence of a security vulnerability. We did not include the failure reports that did not have a security impact on the software system in our study. For this paper, failure and warning densities are calculated by dividing the number of failures and warnings by the number of KLOC (thousands lines of source code) of that component.

8.1.3 Classification of Vulnerabilities According to the CWE

We mapped each vulnerability identified by the failure report to one of 550 weakness classifications of the CWE. The classification provides evidence that the vulnerabilities we have identified in the pre- and post-release failure reports have been known to be vulnerabilities. Additionally, the CWE provides common names to the vulnerabilities that we report which affords the models to be adopted without nomenclature differences between different software engineers.

The CWE contains high- and low-level descriptions of vulnerabilities. For example, the CWE classification (ID) Buffer Errors (119) can include stack overflows and heap overflows. A more specific CWE classification is Stack Overflow (121), which is specific to overflows on the stack and Heap Overflow (122) is specific to overflows on the heap. We mapped the vulnerabilities found in the failure reports to the most specific classification. If not enough information was given in the failure report to distinguish between a high-level and specific CWE classification, then the high-level classification name was assigned to the vulnerability. We assigned the CWE classification identifier Technology-specific Environment Issues (3) to vulnerabilities in our system that were not listed in the CWE. Vulnerabilities in the

Technology-specific Environment Issues (3) included network security vulnerabilities, vulnerabilities specific to the software's design and operation, and absent or weak logging for security audits. Table 4 shows our mapping of vulnerabilities to nine of the ~550 CWE classifications.

Table 4. Vulnerabilities present in the software system in Case Study 1. The ratio of the type of vulnerability to the total number of vulnerabilities is given in the right column.

CWE (ID)	Security vulnerabilities
Resource Exhaustion (400)	32.6%
Technology-specific Environment Issues (3)	32.6%
Null Pointer Dereference (476)	8.7%
Stack Overflow (121)	8.7%
Buffer Errors (119)	6.5%
Insecure Default Permissions (276)	4.3%
Information Leak Through Source Code (540)	2.2%
Permissions, Privileges, and Access Controls (264)	2.2%
Race Conditions (362)	2.2%

FlexeLint was not able to detect all of the vulnerabilities in the system identified by pre- and post-release system test failures. Approximately, 41.3% of the security vulnerabilities identified in the failure reports are not detectable by FlexeLint. That is, FlexeLint contained rules for finding 58.7% of the vulnerabilities in the software system.

8.2 Case Study 2

The second case study was also performed at the anonymous vendor [32]. The system we analyzed consisted of 1,302 C/C++ source files that contributed over one million SLOC. The author and additional research student, doctoral students in software security, analyzed 425 pre-release testing failure records and classified the failures as security-based using the

technique described earlier in Chapter 8.1.2. The vendor's security engineer validated our work. All faults that caused the failures have since been fixed.

The software vendor we partnered with uses Klocwork's static analysis tool to perform automated code analysis. The Klocwork warnings were not audited to filter actual problems from those that were incorrectly flagged as problems (false positives). Thus, the metric, count of Klocwork warnings, includes false positives. We used the warning types as input variables to the model as described in Chapter 8.1.1. We also included Klocwork's coupling metrics, coupling (uses), which measures the relationship between a given file and every other file that file uses. Additionally, we included Klocwork's coupling (used-by) which measures the relationship of each file that uses a given file. Code churn included the count of added, changed, and deleted SLOC.

The failure record analysis results showed that 61/1302 (4.7%) of the source files contained vulnerabilities. The failure analysis showed the following breakdown of vulnerabilities according to the CWE. There were 425 failure reports and 61/425 (14.4%) are classified as vulnerable. Each vulnerability is located in a unique file. The types of failure reports are reported in Table 5.

Table 5. Vulnerabilities present in the software system in Case Study 2. The ratio of the type of vulnerability to the total number of vulnerabilities is given in the right column.

CWE (ID)	Security vulnerabilities
Permissions, Privileges, and Access Controls (264)	3.1%
Insecure Default Permissions (276)	0.2%
Buffer Errors (119)	1.2%
Null Pointer Dereference (476)	0.2%
Stack Overflow (121)	1.2%
Resource Exhaustion (400)	2.6%
Technology-specific Environment Issues (3)	91.2%
Race Conditions (362)	0.22%

8.3 Case Study 3

We performed an empirical case study on a Cisco software system that was implemented in the C programming language [37]. The system is divided into well-defined components upon which our analyses are based. At Cisco, a component is a unit of a software system that typically consists of ten or more files. The count of components is large enough to perform rigorous statistical analyses. We do not provide details of this system due to the proprietary nature of the data. Therefore, we do not report CWE mappings of vulnerabilities in Cisco software or report the types of security-related SCSA warnings.

8.3.1 Candidate Metrics

The first metric is the count and density of warnings produced by two SCSAs that will be called Tool 1 and Tool 2. The names of the tools are not revealed for confidentiality reasons, but these tools are widely used tools in the industry. Cisco requires that SCSA be performed on all components, and validating SCSA warnings as a vulnerability predictor can potentially aid security efforts for all Cisco software. We focus on the security-related SCSA warnings

from Tool 1 to determine if security warnings alone are better predictors of security vulnerabilities than non-security warnings. Examples of Tool 1's security warnings are buffer overflows, poor encryption, permissions problems, and race conditions. We also include null dereferences and memory leaks, which have been documented as causing security problems [59]. We do not restrict our metrics to the security realm, however. Our prior work [31, 36, 39] indicates that non-security failures are positively correlated with security failures, and we therefore do not exclude non-security warnings as candidate metrics. We include non-security warnings from Tool 2, which includes suspicious use of semicolons and ignoring return values of functions. The SCSA warnings in our case study include both true positives and false positives. If the false positives from the SCSA have predictive power in our models, then a time-intensive audit for removing false positives is not required to predict vulnerabilities. By including false positives, we abide by our candidate metric selection criterion that requires that measurements be easily performed.

The second candidate metric for our predictive models is code churn, which is calculated as the sum of added and changed SLOC. In addition to the raw count of code churn, we normalized the churn by KLOC to negate the effect of component size. We suspect that when code is changed in a component, some assumptions about that code or code that is dependent on the changed code may be violated. A vulnerability can be injected into the code when assumptions are made during the code change that are not consistent with the underlying code's assumptions. McGraw [56] uses ambiguity analyses to identify inconsistent assumptions to root out vulnerabilities [56]. Additionally, code churn has been found to be a predictor of fault-prone components [69].

Our third metric is the count of SLOC, which is measured in thousands of SLOC (KSLOC), and can be obtained when calculating code churn. We will test if larger software components are more vulnerable.

The fourth and fifth candidate metrics are the count of faults reported by pre-testing manual static inspections and failures identified by unit testing. The static inspections include faults identified during design reviews and code reviews. These metrics may indicate that faults found early in the SLC are an indicator of faults found later in the SLC. The faults identified by static inspections and unit tests are easily obtainable via queries to the Cisco fault database and are thus included as metrics for our case study. We suspect that as the values of these five metrics grow in a component then so will the count of vulnerabilities.

8.3.2 Security Vulnerabilities

The Cisco Security Evaluation Office (SEO) provided us with the security failures for our case study. The SEO tracks vulnerability trends at Cisco and is thus an optimal source for security data. Software engineers identified these security failures during the testing phase, which includes security testing, stress testing, system testing, and feature/function-level testing. The SEO also provided security failures identified during internal usage and those reported by customers and third-party researchers. These security failures comprise the dependent variable for our predictive models, and will be used to validate if the afore mentioned candidate metrics are good predictors of these security failures. The count and types of security failure reports are not disclosed for confidentiality reasons.

8.4 Attack-prone Components in Case Studies 1, 2, and 3

In this dissertation, an attack-prone component is a component that contains at least one security failure that was identified by testers, customers, or third-party researchers. The attack-prone component is the component that caused the failure and contains the patch that mitigates the vulnerability. We use the term "attack-prone" because the security faults were identified during system execution. In the context of testing, if a tester discovers a buffer overflow during runtime, we say they have attacked the system. Although the tester may not have exploited the buffer overflow to cause a denial-of-service or to inject code that escalates their privileges, the failure is a proof of concept that the system can be attacked. We use the threshold of one security failure to deem a component attack-prone because there is little variability in the security failure count per component and only one attack is needed to cause substantial business loss.

For our analyses, the data that contribute most to the dependent variable are vulnerabilities identified by testers and customers. Therefore, the analyses best predicts what will happen in test as opposed to manual inspections before test. More vulnerabilities identified during manual code inspections may indicate that the models can prioritize manual inspections.

9. Research Methodology

Our predictive models use the candidate metrics to distinguish between attack-prone and not attack-prone components. During model construction, we maximize the predictive model's accuracy, but aim to do so with a small number of included metrics. A larger number of metrics increases the model's complexity and reduces the likelihood that software engineers will adopt the model because more time would be required to gather the metrics, build the model, and incorporate its use in the standard workflow. We first show how effective SCSA warnings are by themselves and compare those results to using multiple additional metrics [28, 29, 32-37]. We also show how effective the models are as compared to random sampling of components for test case prioritization.

The metrics in our predictive models can also be called independent variables or input variables. The dependent variable is the count of vulnerabilities in each component, and this count is used for correlation analyses. In the context of classifying components, attack-prone and not attack-prone are the two response levels. These response levels do not indicate the type or count of vulnerabilities; they only represent if a component is associated with at least one reported vulnerability or not. The models may be useful for predicting which components contain vulnerabilities, where the nature of the vulnerability ranges from abstract design vulnerabilities to relatively simple coding vulnerabilities, such as simple buffer overflows.

9.1 Correlations to Vulnerabilities

We will test if the candidate metrics are correlated with the count of vulnerabilities identified during testing, internal use, and by customers. When the metrics are significantly correlated with vulnerabilities, they are good candidates for the predictive models. During candidate metric selection, we also test the metrics for collinearity. If the metrics are correlated with each other, we have to simplify the model or perform additional analyses to reduce the collinearity.

9.2 Discriminatory Techniques

We use the following three statistical discriminatory techniques in our models: discriminant analysis, logistic regression, and CART. Each technique classifies observations into distinct groups. These techniques have been shown to be successful in classifying fault-prone and not fault-prone modules in the reliability realm [52]. In our setting, we use a dichotomized scheme where components are labeled attack-prone or not attack-prone according to the approach outlined in Chapter 8.4.

The execution of these three statistical techniques examines different combinations, counts, and orders of metrics entered into the model. A good combination of metrics in the model both separates attack-prone and not attack-prone components and maximizes the R^2 value of each model. R^2 is a measure of the variance in the data that is explained by the model [61].

In the CART technique, each leaf of the tree contains a subset of the system components. A single leaf in the CART tree can then be split further into two leaves by including a metric to increase the probability that one leaf will likely contain attack-prone components while the

other leaf will not likely contain attack-prone components. During our CART analysis, each leaf is split into two leaves if and only if the p-value is at or below 0.05. If the p-value is above 0.05, then we are less confident that the groups of components in the two leaves are sufficiently different to warrant the use of separate leaves.

9.3 Model Validation

A ROC curve that has less than 50% of the area under the curve will be rejected, since the model will not have demonstrated a good fit to the data sets. Additionally, we use five-fold cross-validation to determine if the R^2 of the models is accurate. If the R^2 values are consistent, then the variation explained by the predictive model is an accurate description of the model performance. A model with a higher R^2 explains more variance than a model with a low R^2. For example, if the model has an associated R^2 value of 70%, then we would say that the independent variables account for 70% of the response and that the remaining 30% is exampled by another factor.

Since the sample sizes are small for our case studies, we use V-fold cross-validation. V-fold cross-validation is a preferred method to evaluate the results of our statistical analyses as opposed to a method that uses a training, test, and validation set (page 12 and 307) [13]. In the context of CART, the Brieman et al. [13] describe a small sample size containing only a few hundred observations. In the context of thousands of observations, then the training, validation, test set approach is an efficient means to evaluate the results of CART. We used V-fold cross-validation for all three case studies for consistency in our procedures and results. We choose five-fold cross-validation because five has been shown to be an overall

good number [41]. Also, five has been shown to be roughly comparable to V=10 and V=25 in a digit recognition analysis (page 87), but less accurate in a waveform analysis [13].

9.4 Model Efficacy

The models' efficacies are reported in terms of Type I (false positive) and Type II (false negative) error rates. Also, we provide precision, accuracy, and recall values. The techniques that yield the lowest Type I and Type II error rates are then selected for the predictive models. These terms are now defined.

True positive (TP) – a component correctly classified as attack-prone by the model.

False positive (FP) – a component misclassified as attack-prone by the model.

False negative (FN) – a component misclassified as not attack-prone by the model.

True negative (TN) – a component correctly classified as not attack-prone by the model.

We give measures of accuracy to indicate the overall success of the model; measure of precision to indicate how many not attack-prone components are false positives; and measures of recall to indicate how well the model correctly classifies attack-prone components.

$$\text{Accuracy} = \frac{TP + TN}{TP + FP + TN + FN}$$

$$\text{Precision} = \frac{TP}{TP + FP}$$

$$\text{Recall} = \frac{TP}{TP + FN}$$

10.Limitations of the Approach and Threats to Validity of the Case Studies

10.1 Limitations

Case Study 1 and Case Study 3 were performed at the component level. Each component consists of multiple files. As the number of files in a component increases, the less advantageous the prioritization technique becomes. The most effective prioritization is likely to occur at the most granular level (e.g. file or method); prioritizing large components requires all files in the components to be inspected. Additionally, prioritizing for late cycle testing (a largely black box testing activity) is more difficult than white box testing because an inside knowledge of testing individual statements, methods, or control/data flows is required by the tester. Instead gray box testing would have to be performed to peer into the code while testing from the outside of the system. Further, since most of the dependent variable came from late cycle testing, the approach is essentially validated by late cycle tests and no claims with confidence can be made for earlier development testing activities (e.g. unit testing). Lastly, the models presented do not indicate the types of vulnerabilities that are present in the design or code of the system; they only indicate where the vulnerabilities reside in the system. However, these models indicate any kind of vulnerability is present. That is, these models are not limited to specific types of vulnerabilities such as models presented in [74].

10.2 Threats to Validity

Software faults identified by testing failures do not represent the entire set of faults in the software system [22], and customer-reported failures do not completely identify all security faults latent in the system. Additionally, we cannot be certain that all components have been inspected and tested equally. As a result of these uncertainties, our analyses are necessarily based on incomplete data. Adding to this, security issues are rare events in software systems [1]. The scarcity of data makes statistical analyses difficult. For example, the Spearman rank correlation may not perform optimally in zero-inflated data that represent components with no reported security problems. Lastly, the models presented in this paper are representative of only three industrial software systems, and will not necessarily yield the same results on different software systems.

Multiple hypothesis should be investigated whenever performing the scientific method [15]. If only one hypothesis is investigated, then the investigator may result in searching for evidence as opposed to examining a more likely explanation to a question [15]. In this dissertation, there is only one hypothesis – that SCSA warnings are coupled to vulnerabilities identified by testers and those reported in the field. There may be other hypotheses that we should have tested to determine if which components are most susceptible to attack.

According to Campbell [15], "Hypotheses can be eliminated, but not confirmed with absolute certainty…But we can ***never*** prove that [a] hypothesis is the true explanation. It is impossible to repeat an experiment enough times to be absolutely certain that the results will always be the same [15]." In this dissertation, we perform only three empirical case studies. The software systems we investigate are not representative of all software systems and thus

the results may not be consistent with future analyses on the same or other software systems. We do not prove or guarantee anything in this dissertation nor do we make any claims that the results can be used without doubt in any software environment (e.g., academic, industrial). The research presented is strictly empirical with no mathematical proofs that assure answers or results for the software systems investigated or for other software systems.

A best practice in scientific case studies is to report uncertainties in statistical analyses [13]. Another threat to the validity of our results is that we do not report confidence intervals for the probability that each component belongs in a terminal node in the tree. Determining the standard error and confidence intervals for a single observation in CART is difficult enough for a point estimate and is a topic of recent research (e.g., [10]. It is not at all clear how to obtain a rigorously valid standard error estimate for a cross-validation estimate since the random variables are by no means independent (page 307) [13]. However, a heuristic that ignores the lack of independence is available and there are some indications that the estimates work well in practice (page 307-308) [13].

We try to minimize the error by adhering to low P values (below 0.05). Further we kept the trees small with large terminal in the case studies to have the most accurate results (page 255) [13]. The small tree and node size for increasing accuracy is most applicable to regression trees (not to classification trees), but we decided to adhere to this advice in our work with classification trees.

Lastly, there was some subjective interpretation in the analysis of failure reports for case studies 1 and 2 of pre- and post-release failure reports though the cross examination between two doctoral students and one industrial security expert strengthens our results.

11. Results for Three Case Studies

Remember that all models are wrong; the practical question is how wrong do they have to be to not be useful

– George Box

11.1 An Analysis of When Vulnerabilities are Identified

We start by answering Question 1 from Chapter 7. What is the ratio of vulnerabilities identified during the coding phase to those identified post-development? In each case study, the majority of vulnerabilities documented in the fault tracking database were identified during either late cycle system-level testing or in the field as shown in Table 6. The data we analyzed indicated that 100% of the vulnerabilities were identified after the coding phase for both software systems at the anonymous vendor. The results of the same analysis at Cisco are confidential, but there is evidence that an early predictive model can assist Cisco with making more cost-effective security solutions.

Table 6. Percentage of vulnerabilities identified during late stages of the software process.

	Percentage of vulnerabilities found before late-cycle testing.	% of vulnerabilities found during late-cycle testing or reported in the field by customers or third-party researchers
Case study 1	0%	100%
Case study 2	0%	100%
Case study 3	Confidential	Confidential

11.2 Correlations Between SCSA Warnings and Vulnerability Counts for Three Case Studies

The correlations between SCSA warnings and vulnerability accounts are now reported. By definition (see Chapter 2.4.1), all of the correlations measured in the three case studies are weak, although significant at the 95% confidence level or higher (see Table 7). The correlations between SCSA security warnings and vulnerability counts in the three case studies are all approximately 0.2. As mentioned in Section 8.1.2 most of the vulnerabilities for Case Study 1 and Case Study 2 are technology-specific problems and are thus not detectable with the default SCSA rules. The correlation between SCSA warnings and vulnerabilities is a statistical correlation; we do not claim the relationship to be causal.

Table 7. Correlations between SCSA warnings and vulnerabilities for three case studies. All of the observed p-values are at or below 0.05.

SCSA metric	correlation (case study 1)	correlation (case study 2)	Correlation (case study 3)
Null pointer warnings	0.39	0.20	Confidential
Memory leak warnings	--	0.20	Confidential
Buffer overflow	--	0.15	Confidential
Total security warnings	0.42	0.22	0.18
Non-security warnings	--	0.23	0.20
Total warnings	0.40	0.22	0.24
Security warning type 1	--	--	0.19
Security warning type 2	--	--	0.18
Security warning type 3	--	--	0.16

In case studies 2 and 3, the correlations between non-security warnings and vulnerabilities identified during testing is 0.23 and 0.22, respectively. We observe that security warnings by themselves have an almost equal correlation to vulnerabilities as the count of the non-security warnings. The results indicate that non-security problems, a larger

population than security-problems, are also weakly, but significantly, correlated to vulnerabilities. Therefore, the discirmination between security-related warnings and non-security-related warnings is not required to identify components that have other exploitable vulnerabilities. The remaining subsections present a meaningful description of what "weak, but significant" implies in the context of a correlation between SCSA warnings and vulnerabilities.

11.3 CART with SCSA only for Three Case Studies

We observe that the attack-prone components are distributed more to the left leaves of the CART trees than the right leaves. That is, the components in the leftmost leaves have a higher probability of being attack-prone than components in the rightmost leaves. Therefore, we can rank the components by probability of being attack-prone by starting with the leftmost leaves and traversing the leaves to the rightmost leaf. Software engineers can normalize the split values in their trees and apply them to the model for the next release of the software system.

We show the percent of attack-prone components in the (approximately) top ten percent (the leftmost leaves that constitute ten percent of the components) and the (approximately) top 20% of the components probability ranking in Tables 8 and 9. We cannot precisely identify the top ten percent and 20% exactly due to the differing number of components in the leaves and so the percentages in Tables 8 and 9 are not exactly 10% and 20%. In Case Study 1, the left most leaf represented 28% of the components in the system and thus we do consider a recall of 10%.

Table 8. Model efficacy for SCSA warnings alone when considering the top 10% of software components.

	Accuracy	**Precision**	**Recall in top x%**
Case study 1	--	--	--
Case study 2	90.2%	25.4%	55.7% in top 10.3%
Case study 3	83.0%	36.2%	36.2% in top 14.8%

Done by looking at the leaves closest to 10%. First row not possible around 10%.

In Table 9, for Case Study 3, a 20% threshold is not possible due to the arrangement of components and leaves in the leftmost portion of the tree.

Table 9. Model efficacy for SCSA warnings alone when considering the top 20% of software components.

	Accuracy	**Precision**	**Recall in x%**
Case study 1	40.0%	100%	70.0% in top 28%
Case study 2	83.7%	16.5%	60.6% in top 17.2%
Case study 3	--	--	--

The results indicate that a substantial portion of attack-prone components can be identified in the leftmost leaves for case studies 1, 2, and 3 with SCSA warnings as the only input variable (metric) to the model. For example, 60.6% of the attack-prone components in the top 17.2% of the component ranking in Case Study 2. The recalls at 10% and 20% in Tables 8 and 9 provide meaningful implications to the weak, but statistically significant correlations. We observe a substantial portion of the attack-prone components can be identified, but given the low precisions of the model, software engineers may prefer alternative techniques or require the models to be enhanced before adopting such a model in their production environments.

We now compare the model's predictive power to random sampling of the software components. The random sampling represents how many attack-prone components software engineers would analyze in the top 10% and 20% of the software system if they did not use the predictive model. We take five random samples of the components and average the successful identification of attack-prone components during the random sampling as shown in Tables 10 & 11. A comparison between the identifying attack-prone components with the models and random sampling demonstrates the efficacy of the model. For example, the model for Case Study 1 at the 20% threshold is 70.0% while 26.0% with random sampling, a 169.2% increase in attack-prone component identification.

Table 10. Efficacy of finding attack-prone components via random sampling 10% of the software components.

Recall rate SCSA only	**Random sample 1**	**Random sample 2**	**Random sample 3**	**Random sample 4**	**Random sample 5**	**Average**
Case study 1 at top 10%	--	--	--	--	--	--
Case study 2 at top 10.3%	8.2%	8.2%	6.6%	11.5%	11.5%	9.2%
Case study 3 at top 14.8%	40.9%	44.3%	47.5%	37.7%	36.1%	41.3%

Table 11. Efficacy of finding attack-prone components via random sampling 20% of the software components.

	Random sample 1	**Random sample 2**	**Random sample 3**	**Random sample 4**	**Random sample 5**	**Average**
Case study 1 at top 28%	30.0%	10.0%	30.0%	20.0%	40.0%	26.0%
Case study 2 at top 17.2%	21.3%	19.7%	18.0%	14.8%	9.8%	16.72%
Case study 3 at top 20%	--	--	--	--	--	--

Components with many security-related SCSA warnings are *vulnerability*-prone components (as opposed to *attack*-prone components) due to the static identification of the vulnerabilities. The components that are most vulnerability-prone may also be attack-prone if the vulnerabilities are likely to be exploited. The vulnerabilities can be a combination of the SCSA warnings that have security impact and the other types of vulnerabilities present in that component. The correlations indicate that a vulnerability-prone component, classified as vulnerability-prone due to the presence of a large population of SCSA faults, is also an attack-prone component.

12. Results for Three Case Studies with Multiple Metrics

We now repeat the analyses in Chapter 11, but we now use multiple metrics for each case study. The metrics for the case studies are given in Table 12. The same metrics were not available for each case study. The coupling metric was only available in Case Study 2. Also, for Case Study 3, the count of faults found by manual inspections was available, but not in the other two case studies.

Table 12. Metrics for the three case studies

	Metrics (count per component)
Case study 1	SCSA warnings, churn
Case study 2	SCSA warnings, churn, coupling
Case study 3	SCSA warnings, size, churn, manual inspections

12.1 Correlation between Candidate Metrics and Vulnerability Counts for Three Case Studies

We performed Spearman Rank correlations for the following additional metrics added to the model: code churn and size, coupling, and the count of faults found manually. All correlations are provided in Table 13. The correlation between code churn and the count of vulnerabilities was among the strongest correlations for each case study. The size of the software component is also correlated to the count of vulnerabilities, but the low correlation indicates that size is not the only factor for vulnerabilities. All correlations in Table 13 have a p-value at or below 0.05 (95% significance level).

Table 13. Spearman rank correlation coefficients between metrics and vulnerability counts. N/A indicates the metric was not available for the data set.

Metric	Case study 1	Case study 2	Case study 3
Code churn	0.4	0.4	0.2
Size (SLOC)	0.4	0.3	0.2
Coupling	N/A	0.2	N/A
Count of faults found manually	N/A	N/A	0.2

12.2 Collinearity between Candidate Metrics for Three Case Studies

We tested the collinearity between the different candidate metrics. In Case Study 1, there were only two metrics used, alert density and churn. We observed no correlation at the .05 level or below for these metrics. The correlations between metrics in Case Study 2 are presented in Table 14. The strongest correlation was 0.4 and existed between churn and coupling. The strength of this correlation was not strong enough to prevent the use of both metrics in the same model. In Case Study 3, the largest correlation coefficient is 0.56 and exists between KSLOC and all SCSA, as shown in Table 15. These candidate metrics were not used together in the same predictive model because the correlation indicates that they describe approximately the same phenomena in the data set. The other correlation coefficients are weak and were therefore not considered a threat to validity. All correlations in Table 14 and 15 have a p-value at or below 0.05 (95% significance level).

Table 14. Correlation matrix for candidate metrics for Case Study 2

Metric	SCSA warnings	Churn	Coupling
SCSA warnings	--	0.26	0.36
Churn	--	--	0.42
Coupling	--	--	--

Table 15. Correlation matrix for candidate metrics for Case Study 3.

Metric	SCSA Warnings	KSLOC	Churn	Static inspections	Unit tests
SCSA warnings	--	0.56	0.25	0.40	0.16
KSLOC	--	--	0.52	0.40	0.35
Churn	--	--	--	0.34	0.40
Static inspections	--	--	--	--	0.38
Unit tests	--	--	--	--	--

12.3 CART Models with Multiple Metrics for Three Case Studies

The rankings of the attack-prone components for the top ten percent and 20% level are given in Tables 16 and 17. In comparison to SCSA metrics as the sole predictor in a model, the multiple metrics increase the identification by 42.9% for Case Study 1 and 48.8% for Case Study 2 when examining the top 20%. In Case Study 1 and Case Study 2, we analyze the top 28% and top 17.2% when a single metric in the model is considered, but when the metrics are combined we analyze the top 20.0% and 18.6%. The percentages are not equivalent due to the count of the components in the leftmost leaves, however the percentages are similar enough to indicate that in our case studies, multiple metrics provided more predictive power than a single predictor. The trees produced by the models are shown in Appendix A, B, C for Case Studies 1, 2, and 3, respectively.

Table 16. Model efficacy for multiple metrics when considering the top 10% of software components.

	Accuracy	**Precision**	**Recall in top x%**
Case study 1	--	--	--
Case study 2	91.6%	32.4%	72.1% in top 10.4%
Case study 3	93.3%	74.1%	48.7% in top 8.5%

Table 17. Model efficacy for multiple metrics when considering the top 20% of software components.

	Accuracy	**Precision**	**Recall in x%**
Case study 1	88.0%	80.0%	100% in top 48.0%
Case study 2	83.8%	21.2%	90.2% in top 20.0%
Case study 3	88.0%	52.5%	75.6% in top 18.6%

We performed a random sampling of components as performed in Chapter 11.3. The predictive model identifies 446.7% more attack-prone components for Case Study 2 and 332.0% more for Case Study 3 when assessing the top 20% of the probability as shown in Tables 18 and 19. Therefore, the predictive with multiple metrics demonstrate a higher utility over random sampling that would occur when software engineers select components to perform security inspections on.

Table 18. Efficacy of finding attack-prone components via random sampling 10% of the software components.

Recall rate SCSA only	**Random sample 1**	**Random sample 2**	**Random sample 3**	**Random sample 4**	**Random sample 5**	**Average**
Case study 1 at top 10%	--	--	--	--	--	--
Case study 2 at top 10.4%	3.3%	16.4%	13.1%	9.8%	11.5%	10.8%
Case study 3 at top 8.5%	2.4%	12.2%	14.6%	9.7%	7.3%	9.24%

Table 19. Efficacy of finding attack-prone components via random sampling 20% of the software components.

Casey study SCSA only	**Random sample 1**	**Random sample 2**	**Random sample 3**	**Random sample 4**	**Random sample 5**	**Average**
Case study 1 at top 48.0%	50.0%	60.0%	50.0%	60.0%	60.0%	56.0%
Case study 2 at top 20.0%	21.3%	24.6%	18.0%	19.7%	18.0%	16.5%
Case study 3 at top 18.6%	14.6%	14.6%	29.3%	14.6%	14.6%	17.5%

Based on the results of the models with multiple metrics, we say that "weak, but significant" is a technically accurate description for SCSA warnings, but that a significant correlation alone should not provide enough justification to accept a result. We have shown empirical evidence of multiple metrics that have weak, but significant correlations are good predictors of attack-prone components.

12.4 Discussion of CART Results

We observed that the attack-prone components are most likely to be in the leftmost leaves of the trees while the remaining attack-prone components are scattered throughout the remaining leaves (that have high Type I error rates). The CART tree makes a general ranking – in terms of probability that the component in the leaf is attack-prone. We can rank the components by probability that they are attack-prone by starting with the leftmost leaf and traversing rightward in the tree. Note well that the probabilities of the components in the leaves are probabilities that indicate the chances that selecting a component in that leaf is attack-prone. The table view of the CART analysis is provided in Table 20.

Table 20. CART results for Case Study 3.

Leaf No.	Conditional probability formula	Probability not attack-prone	Probability attack-prone
1	SecurityWarningType4>=a&Tool1CountWarnings>=b&Tool2WarningDensity>=c& SecurityWarningType4Density<d	0.00	1.00
2	SecurityWarningType4>=a&Tool1CountWarnings>=b&Tool2WarningDensity>=c&SecurityWarningType4Density>=d&DesignAnalysisFaults>=e&AllManualStaticInspections<f&SecurityWarningType2>=g	0.00	1.00
3	SecurityWarningType4>=a&Tool1CountWarnings>=b&Tool2WarningDensity>=c&SecurityWarningType4Density>=d&DesignAnalysisFaults>=e&AllManualStaticInspections<f&SecurityWarningType2<g	0.53	0.46
4	SecurityWarningType4>=a&Tool1CountWarnings>=b&Tool2WarningDensity>=c&SecurityWarningType4Density>=d&DesignAnalysisFaults>=e&AllManualStaticInspections>=f	1.00	0.00
5	SecurityWarningType4>=a&Tool1CountWarnings>=b&Tool2WarningDensity>=c&SecurityWarningType4Density>=d&DesignAnalysisFaults<e&Security WarningType3<h	0.50	0.50
6	SecurityWarningType4>=a&Tool1CountWarnings>=b&Tool2WarningDensity>=c&SecurityWarningType4Density>=d&DesignAnalysisFaults<e&Security WarningType3>=h&Tool2WarningCount>=i	0.62	0.37
7	SecurityWarningType4>=a&Tool1CountWarnings>=b&Tool2WarningDensity>=c&SecurityWarningType4Density>=d&DesignAnalysisFaults<e&Security WarningType3>=h&Tool2WarningCount<i	0.98	0.01
8	SecurityWarningType4>=a&Tool1CountWarnings>=b&Tool2WarningDensity<c	1.00	0.00
9	SecurityWarningType4>=a&Tool1CountWarnings<b	0.97	0.02
10	SecurityWarningType4<a&Churn>=j&Churn<k	0.50	0.50
11	SecurityWarningType4<a&Churn>=j&Churn>=k	0.93	0.06
12	SecurityWarningType4<a&Churn<j	1.00	0.00

The first six leaves of the tree contained a substantial portion of attack-prone components; the remaining six leaves did attack-prone components, but also contained many not attack-prone components, too. We provide a visualization of the results from the sequential tree analysis in Figure 2.

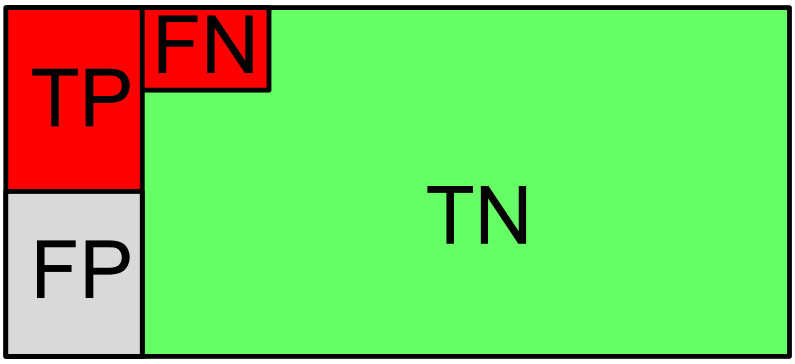


Figure 2. Visualization of model results for Case Study 3 after the CART analysis.
TN (True Negatives - correctly classified as not attack-prone)
FN (False Negatives - misclassified as not attack-prone)
TP (True Positives - correctly classified as attack-prone)
FP (False Positives - misclassified as attack-prone)

13. Model Validation for Three Case Studies

13.1 ROC Curves and Cross-validation

We evaluated the models using ROC curves and five-fold cross-validation. Table 21 provides the area under the curve (AUC) for the CART models with SCSA warnings alone and Table 21 provides the AUC for multiple metrics. The AUC for the three case studies is more consistent with multiple metrics than with SCSA warnings alone indicating that the CART predictions are more consistent and thus give more confidence in their value as shown in Figures 4-9. The model's false positive rate is along the x-axis and the true positive rate (sensitivity) is along the y-axis. The blue curve is a representation of the sorting efficiency of the estimated probabilities that a component is attack-prone. When the model successfully identifies an attack-prone component, then the blue curve moves upward by one unit. Each time the model classifies an attack-prone component that was not associated with an attack (i.e. a false positive), then the blue curve moves one unit right along the x-axis. The model identifies all of the attack-prone components when the blue curve reaches 1.00 on the y-axis. The gray diagonal line serves as a baseline, where the false positive rate is equal to the true positive rate to represent a model that has no predictive power. The red curve is the inverse of the blue curve and represents how well the model is at identifying not attack-prone components. The ROC curve for each model has over 92% of the area under the curve (AUC), indicating that the model demonstrates a good fit to the data sets. The consistency between the ROC curves indicates that the model is an appropriate model for predicting

attack-prone components and provides evidence that the predictions may be successful for other software systems.

Table 21. Area under the curve (AUC), R^2 value, and cross-validated R^2 when considering the top 20% of the software components in models with SCSA warnings alone.

	AUC	**R^2**	**Cross-validated R^2**
Case study 1	0.930	0.637	0.504
Case study 2	0.783	0.273	0.246
Case study 3	0.700	0.203	0.180

Table 22. Area under the curve (AUC), R^2 value, and cross-validated R^2 when considering the top 20% of the software components in models with multiple metrics.

	AUC	**R^2**	**Cross-validated R^2**
Case study 1	93.0%	67.9%	61.1%
Case study 2	91.9%	42.1%	38.2%
Case study 3	94.4%	55.6%	50.4%

We observe that the AUC is inconsistent when SCSA alone is used. However, the AUC produced from models with multiple metrics consistent. The consistency indicates more confidence that the model can be used on different software systems and produces the similar ratio of true positives and false positives. However, when looking at only the top 10% or 20% of the components in the software system, we are only assessing the left most portion of the ROC curve which can be generally described as having a large and positive slope. The leftmost leaves generate the probability ranking in Tables 8, 9, 16, and 17. We conclude that the answer to Question 3 is that additional metrics create a more stable model than a model that uses SCSA warnings alone.

The cross-validated R^2 values (see Tables 21 and 22) indicate that not all of the response in the data is accounted for, but the overall R^2 values are fairly consistent with the cross-

validated R^2 values. Similar to the AUC analysis, the R^2 values and the cross-validated R^2 values are erratic. The range of the R^2 for models with SCSA warnings alone is 43.4%, while 25.8% for models with multiple metrics. While the R^2 values are modest, the splits in the leaves of the CART models was performed at or below the 95% confidence level.

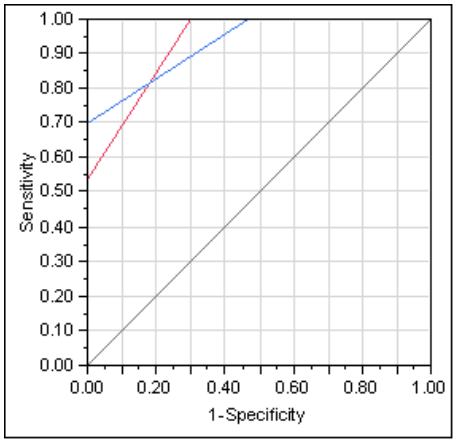


Figure 3. Case Study 1 ROC curve. SCSA only.

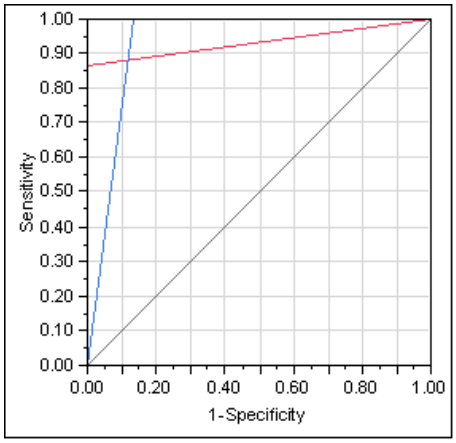


Figure 4. Case Study 1 ROC curve. Multiple metrics.

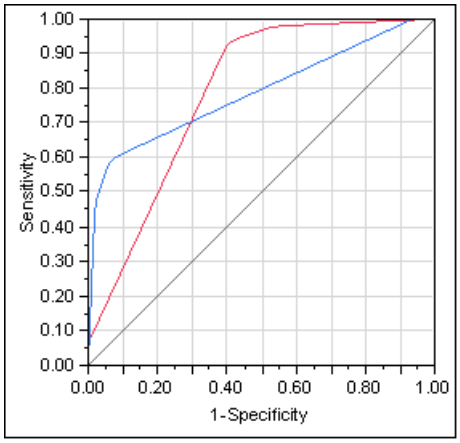


Figure 5. Case Study 2 ROC curve. SCSA only.

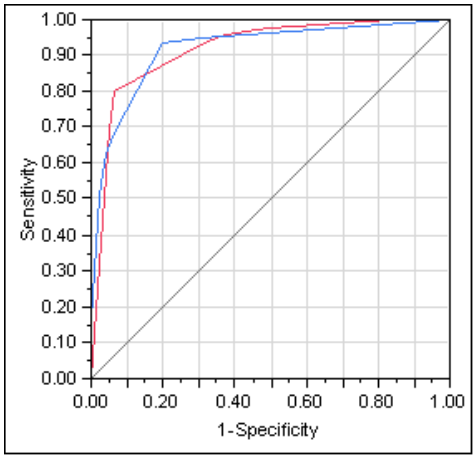


Figure 6. Case Study 2 ROC curve. Multiple metrics.

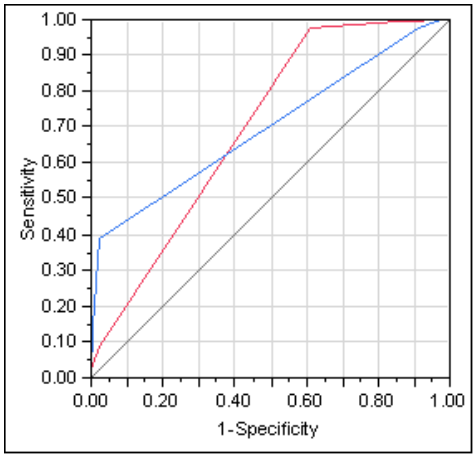


Figure 7. Case Study 3 ROC curve. SCSA only.

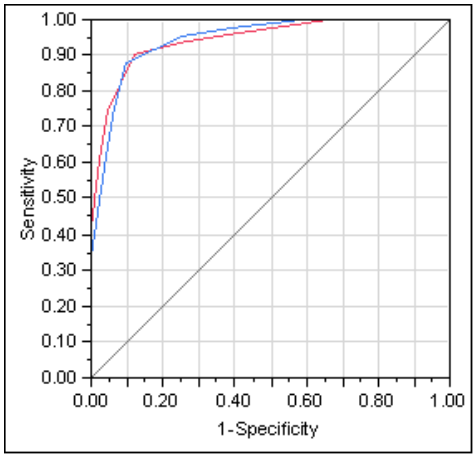


Figure 8. Case Study 3 ROC Curve. Multiple metrics

13.2 Cost Analysis of Models

The ROC curve of predictive model that identifies all false positives after identifying all true positives would have the curve start at the origin and traverse vertically along the y-axis. When the curve stops at 1.00, the maximum value of the y-axis, the curve would traverse horizontally rightward along the x-axis until the false positive rate was 1.00. A cost analysis of such a model would indicate that software engineers should inspect all components until they inspect the first false positive. While our models do not have perfectly predict the attack-prone components, we can see there is still value to the models. In Figure 6, we see that the blue curve starts at the origin and traverse vertically along the y-axis until approximately 40% of the attack-prone components are identified. A cost analysis for the model indicates that the first false positive is not inspected until 40% of the attack-prone components are fortified. In Figure 4, the model has perfect prediction until 70% of the attack-prone components are predicted and then the false positive rate becomes non-zero. Based on these predictions, software managers can determine how many resources to allocate to the identification and fortification of vulnerabilities in the attack-prone components. If all attack-prone components are required to be assessed, then the manager can use the ROC curve to perform a cost analysis of how many components software engineers should inspect to achieve 100% attack-prone component remediation.

13.3 Contribution of SCSA Warnings in the CART Models with Multiple Metrics

The predictive power of the metrics is measured by the likelihood-ratio chi-square statistic, G^2, for each metric. A larger G^2 value indicates a better statistical fit, which indicates a better split of the leaves in the CART analysis between attack-prone and not attack-prone components. During the analysis, the total G^2 is provided and represents the total possible contribution of the metrics to the model. The column contributions then give the G^2 for each metric. The R^2 is calculated by dividing the sum of the G^2 values for the metrics divided by the total G^2 value possible.

In Case Study 3, the total number of SCSA warnings metric contributed the most separation in the overall model, as shown in Table 23. For Case Study 1, the contribution of warnings was almost equal to churn. In Case Study 2, warnings had the second most contribution to the model. These results show that the faults identifiable by SCSA are predictive of different types of security faults that surface during testing and those reported in the field.

Table 23. G^2 values for models with multiple metrics.

	SCSA warnings	**Churn**	**File coupling**	**Static inspections**
Case study 1	10.6	12.2	--	--
Case study 2	32.2	156.6	18.6	--
Case study 3	76.1	24.9	--	20.2

Based on the positive and significant correlations, the statistically significant splits in the CART analyses, the cross-validated R^2 values, the ROC tables, and the G^2 values, we make the following theories:

Theory: *Above a statistically determined threshold, SCSA vulnerability warnings are in the same components as vulnerabilities that are likely to be exploited*.

Although the correlations are weak (but statistically significant), the CART models indicate that when SCSA warnings are of high counts in the same components as large values of other metrics, the warnings are coupled to other types of vulnerabilities. Therefore, we theorize that SCSA fault-dynamic coupling effect is a plausible statistical statement that is observed in our three case studies. In other words, software engineers will find vulnerabilities not identifiable by SCSA in the same components as SCSA warnings.

14. Discussion

We have provided evidence of statistical correlations between SCSA warning counts and vulnerability counts. We provide some justifications for why this correlation exists. McGraw [56] suggests that an architectural risk analysis can reveal vulnerabilities when ambiguities are present in product. For example, an ambiguity in a design document may reveal a design flaw (perhaps stemming from ambiguous requirements). We suspect that requirements may have been ambiguous, which led to designers drafting an ambiguous design, which led to developers implementing ambiguous (and thus vulnerable) code. Therefore, the vulnerabilities identifiable by SCSAs may be in the same components where the higher design-level flaws exist.

The SCSAs generated false positives, but our results indicate that false positives still have predictive power. Although the code flagged as faulty by the SCSA may not cause a failure, the SCSAs classified the code as faulty because the code closely resembles the definitions of actual faults that the SCSAs use to find faults in source code. The more faulty the code appeared to the tool, the more likely the component contains an actual vulnerability, according to our findings. Further, the results suggest that seemingly faulty code is in the same component as the actual faulty code. Therefore, these components contain less than desirable coding standards.

Despite the weak correlations, the predictive power of these metrics can be used as a guide. This observation is consistent with the idea that "…a seemingly worthless split might lead to a very good split below it" [41]. In our earlier study [36], we found that weakly-correlated metrics have discriminatory power equal to strongly-correlated non-security

failures. The weaker metrics have less significant splits, although the p-values are below 0.05, than the strongly-correlated metric. However, our observations in [36], and in comparison of the results presented here with those in [39], we find that weakly-correlated metrics can lead to more splits than strongly-correlated metrics, and that many splits can yield good separation between attack-prone and not attack-prone components.

15. Summary of Research Contributions, Observations, and Theories

The goal of this research is to reduce vulnerabilities from escaping into the field by incorporating source code analyzer warnings into statistical models that predict the presence of more complex vulnerabilities. The hypothesis for our research is that SCSA warnings are in the same component as more complex coding vulnerabilities and vulnerabilities associated with the design and operation of the software system.

Observation: False positives from source code static analyzers are useful for predicting attack-prone components; audits of source code static analyzers are not required for predictions.

Observation: A source code static analyzer warning does not need to be security-related to predict a vulnerability.

Observation: static and dynamic metrics can have similar predictive powers in the context of source code static analysis tools and vulnerabilities identified during program execution.

We have demonstrated three empirical case studies that indicate that SCSA warnings can identify components that contain vulnerabilities identified by late-cycle testing. The correlations are positive and statistically significant. Further, during the CART analysis, all splits of the tree were performed with p-values at or below 0.05. The models shown separate the attack-prone components from not attack-prone components with significance, and therefore we do not reject the alternative hypothesis. Additionally, the AUCs are consistent providing some indicates that the models have consistent fit across three large commercial

telecommunications systems from two different vendors. Based on these findings, we state the following theory:

Theory: Above a statistically determined threshold, SCSA vulnerability warnings are in the same components as vulnerabilities that are likely to be exploited. In other words, software engineers will find vulnerabilities not identifiable by SCSA in the same components as SCSA warnings.

The models reduce the search space for security efforts to identify vulnerable components. Security engineers can prioritize their efforts to the attack-prone components to mitigate potential damage from the highest risk components.

16. Future Work

The CART models presented in this dissertation are not dissimilar to those in our other case studies [36, 38, 39]. The CART models showed that the best single indicator of security faults and failures were those components with the most non-security failures as documented in [31, 36, 39]. In Case Study 3, the components with the most customer-reported failures had the most security failures. This observation suggests that the customers' operational profiles may influence vulnerable execution flows in the software that were not identified during pre- or post-release testing. We conclude that non-security failures and the predictors of non-security failures are potential metrics security-related predictive models. Given that reliability and security problems exist in the same components, it may be worthwhile to unify the concepts of "software reliability engineering" and "software security engineering" into a single theme (e.g. software trustworthiness engineering) to indicate that security and reliability folks should collaborate in the same sections of the software system and that security should be kept in mind when the system becomes unreliable. We carefully choose the term *trustworthiness* according to Bishop [8]. We also assume that security is a subset of reliability and thus assurance is a subset of trustworthiness. Additionally, the correlation between reliability and security failures may indicate that the extensive research on reliability statistical models [51, 77] that have been shown to predict fault- and failure-prone components early in the SLC may also be helpful for security prediction models. Next, we will examine if the non-security failures in our dataset can actually afford an attacker a means to exploit the software system. The initially classified non-security failures may

provide opportunities for new types of security attacks. These security holes may result from not abiding by the principle of fail-safe defaults because the failure to perform the required functions within the specified performance requirements opened a security hole. Fail-safe defaults is a security design principle [82].

REFERENCES

[1] Alhazmi O. H., Y. K. Malaiya, and I. Ray, "Measuring, analyzing and predicting vulnerabilities in software systems," *Computers & Security*, vol. 26, no. 3, pp. 219-228, May 2006.

[2] Anderson J., "Computer Security Technology Planning Study," Fort Washington, October 1972.

[3] Barnum S. and M. Gegick, "Design Principles," https://buildsecurityin.us-cert.gov/portal/article/knowledge/Principles, 2005.

[4] Basili V., G. Caldiera, and R. D. Rombach, *Goal Question Metric Approach*, John Wiley & Sons, Inc., 1994.

[5] Basili V., L. Briand, and W. Melo, "A Validation of Object Oriented Design Metrics as Quality Indicators," *IEEE Transactions on Software Engineering*, vol. 22, no. 10, pp. 751-761, 1996.

[6] Basili V., F. Marotta, K. Dangle, L. Esker, and I. Rus, "Measures and Risk Indicators for Early Insight Into Software Safety," *CrossTalk: The Journal of Defense Software Engineering*, vol. 21, no. 10, pp. 4-8, October 2008.

[7] Binkley A. B. and S. Schach, "Validation of the coupling dependency metric as a predictor of run-time failures and maintenance measures," *International Conference on Software Engineering*, Kyoto, Japan, pp. 452-455, 1998.

[8] Bishop M., *Computer Security: Art and Science*, Boston, Addison-Wesley, 2003.

[9] Biyani S. and P. Santhanam, "Exploring defect data from development and customer usage on softare modules over multiple releases," *International Symposium on Software Reliability Engineering*, Paderborn, Germany, pp. 316-320, 1998.

[10] Bloch D. A., R. A. Olshen, and M. G. Walker, "Risk estimation for classification trees," *Journal of Computational and Graphical Statistics*, vol. 11, pp. 263-268, 2002.

[11] Boehm B., *Software Engineering Economics*, New Jersey, Prentice-Hall, 1981.

[12] Boehm B., *Software Risk Management*, Washington, IEEE Computer Society Press, 1989.

[13] Breiman L., J. Friedman, R. Olshen, and C. Stone, *Classification and Regression Trees*, Boca Raton, Chapman & Hall/CRC, 1984.

[14] Briand L., J. Wust, and H. Lounis, "Investigating Quality Factors in Object-Oriented Designs: An Industrial Case Study," *International Conference on Software Engineering*, Los Angeles, California, pp. 345-354, May 1999.

[15] Campbell N., *Biology*, Redwood City, CA, The Benjamin/Cummings Publishing Company, Inc, 1987.

[16] Chandra P., B. Chess, and J. Steven, "Putting the Tools to Work: How to Succeed with Source Code Analysis," IEEE Security & Privacy, vol. 4, no. 3, pp. 80-83, May/June, 2006.

[17] Chess B. and J. West, *Secure Programming with Static Analysis*, Boston, MA, Addison Wesley, 2007.

[18] Crawford S. G., A. A. McIntosh, and D. Pregibon, "An analysis of static metrics and faults in C software," *Journal of Systems and Software*, vol. 5, no. 1, pp. 37-48, February 1985.

[19] DeMillo R. A., R. J. Lipton, and F. G. Sayward, "Hints on test data selection: Help for the practicing programmer," IEEE Computer, vol. 11, no. 4, pp. 34-41, April, 1978.

[20] Denaro G., "Estimating software fault-proneness for tuning testing activities," *International Conference on Software Engineering*, St. Malo, France, pp. 269-280, 2000.

[21] Devore J. L., *Probability and Statistics for Engineering and the Sciences*, Belmont, CA, Thomson, 2008.

[22] Dijkstra E., *Structured Programming*, Brussels, Belgium, 1970.

[23] Dowd M., J. McDonald, and J. Schuh, *The Art of Software Security Assessment: Identifying and Preventing Software Vulnerabilities*, Boston, MA, Addison-Wesley, 2007.

[24] El-Emam K., S. Benlarbi, N. Goel, and S. N. Rai, "The Confounding Effect of Class Size on the Validity of Object-Oriented Metrics," *IEEE Transactions on Software Engineering*, vol. 27, no. 7, pp. 630-650, July 2001.

[25] Elbaum S. and J. Munson, "Code Churn: A Measure for Estimating the Impact of Code Change," *International Conference Software Maintenance*, 24-31 November 1998.

[26] Endres A. and R. D. Rombach, *A Handbook of Software and Systems Engineering*, Harlow, England, Pearson Education, Limited, 2003.

[27] Freund R., R. Littell, and L. Creighton, *Regression Using JMP*, Cary, NC, SAS Institute, Inc., 2003.

[28] Gegick M., "Modeling Automated Static Analyzer Alerts to Identify and Predict Vulnerability- and Attack-prone Components," *2nd International Doctoral Symposium at Empirical Software Engineering and Measurement*, Madrid, Spain, September 2007.

[29] Gegick M. and L. Williams, "Correlating Automated Static Analysis Alert Density to Reported Vulnerabilities in Sendmail," *MetriCon 2.0 at 16th USENIX Security Symposium (Security '07)*, Boston, MA, August 2007.

[30] Gegick M. and L. Williams, "Toward the Use of Static Analysis Alerts for Early Identification of Vulnerability- and Attack-prone Components," *First International Workshop on Systems Vulnerabilities (SYVUL '07)* Santa Clara, CA, July 1-6 2007.

[31] Gegick M., "Failure-prone Components are also Attack-prone Components," *OOPSLA - ACM student research competition*, Nashville, Tennessee, pp. 917-918, October 2008.

[32] Gegick M. and L. Williams, "STUDENT PAPER: Ranking Attack-prone Components with a Predictive Model," *ISSRE*, Redmond, WA, pp. 315-316, November 2008.

[33] Gegick M. and L. Williams, "Predicting Where Software Systems will be Attacked," *High Confidence Systems and Software*, Linthicum Heights, MD, 7 March 2008.

[34] Gegick M. and L. Williams, "Internal and External Metrics for Predicting Attack-prone Components," *MetriCon2.5*, San Francisco, CA, 7 April 2008.

[35] Gegick M., L. Williams, J. Osborne, and M. Vouk, "Prioritizing Software Security Fortification through Code-Level Security Metrics," *Workshop on Quality of Protection*, Alexandria, VA, pp. 31-37, October 2008.

[36] Gegick M., L. Williams, and M. Vouk, "Predictive Models for Identifying Software Components Prone to Failure During Security Attacks," Build Security In (https://buildsecurityin.us-cert.gov/daisy/bsi/home.html) 2008.

[37] Gegick M., P. Rotella, and L. Williams, "Predicting Attack-prone Components," *International Conference on Software Testing, Verification, and Validation*, Denver, CO, pp. 181-190, April 2009.

[38] Gegick M., P. Rotella, and L. Williams, "Predicting Attack-prone Components," NCSU, Raleigh, TR-2009-1, 26 January 2009.

[39] Gegick M., P. Rotella, and L. Williams, "Toward Non-security Failures as a Predictor of Security Faults and Failures," *ESSoS*, Leuven, Belgium, pp. 135-149, February 4-6 2009.

[40] Gyimothy T., R. Ference, and L. Siket, "Empirical Validation of Object-Oriented Metrics on Open Source Software for Fault Prediction," *IEEE Transactions on Software Engineering*, vol. 31, no. 10, pp. 897-910, October 2005.

[41] Hastie T., R. Tibshirani, and J. H. Friedman, *The Elements of Statistical Learning*, New York, Springer, 2001.

[42] Hoglund G. and G. McGraw, *Exploiting Software*, Boston, Addison-Wesley, 2004.

[43] Howard M. and D. LeBlanc, *Writing Secure Code*, 2nd ed. Redmond, Microsoft Corporation, 2003.

[44] Howard M. and S. Lipner, *The Security Development Lifecycle*, Redmond, Microsoft Press, 2006.

[45] Hudepohl J., S. J. Aud, T. Khoshgoftaar, E. B. Allen, and J. Mayrand, "Emerald: Software Metrics and Models on the Desktop," *IEEE Software*, vol. 13, no. 5, pp. 56-59, September 1996.

[46] Hudepohl J., W. Jones, and B. Lague, "EMERALD: A Case Study in Enhancing Software Reliability," *Eighth International Symposium on Software Reliability Engineering (Case Studies)*, Albuquerque, New Mexico, pp. 85-91, 1997.

[47] IEEE, "ANSI/IEEE Standard Glossary of Software Engineering Terminology (IEEE Std 610.12-1990)," IEEE Computer Society Press, Los Alamitos, CA, 1990.

[48] ISO, "ISO/IEC DIS 14598-1 Information Technology - Software Product Evaluation - Part 1: General Overview," October 28 1996.

[49] ISO/IEC 24765, "Software and Systems Engineering Vocabulary," 2006.

[50] Khoshgoftaar T. and J. Munson, "Predicting Software Development Errors using Software Complexity Metrics," *IEIEE Journal on Selected Areas in Communications*, vol. 8, no. 2, pp. 253-261, February 1990.

[51] Khoshgoftaar T. M., E. B. Allen, K. S. Kalaichelvan, N. Goel, J. P. Hudepohl, and J. Mayrand, "Detection of Fault-Prone Program Modules in a Very Large Telecommunications System," *Sixth International Symposium on Software Reliability Engineering*, pp. 24-33, 1995.

[52] Khoshgoftaar T. M., E. B. Allen, and J. Deng, "Using Regression Trees to Classify Fault-Prone Software Modules," *IEEE Transactions on Reliability*, vol. 51, no. 4, pp. 455-562, December 2002.

[53] Krsul I., "Software Vulnerability Analysis," PhD Thesis in Computer Science at Purdue University, West Lafayette 1998.

[54] Littell R., W. Stroup, and R. Freund, *SAS for Linear Models, Fourth Edition*, Cary, NC., SAS Institute, Inc., 2002.

[55] McCabe T. J., "A Complexity Measure," *IEEE Transactions on Software Engineering*, vol. 2, no. 4, pp. 308-320, 1976.

[56] McGraw G., *Software Security: Building Security In*, Boston, Addison-Wesley, 2006.

[57] McGraw G., "Silver Bullet Podcast - Show 003: An Interview with Marcus Ranum," 14 July 2006.

[58] Menzies T., J. Greenwald, and A. Frank, "Data Mining Static Code Attributes to Learn Defect Predictors," *IEEE Transactions on Software Engineering*, vol. 33, no. 1, pp. 2-13, January 2007.

[59] MITRE, "Common Weakness Enumeration," http://cwe.mitre.org/, 2006.

[60] Morell L. J., "A theory of fault-based testing," *IEEE Transactions on Software Engineering*, vol. 16, no. 8, pp. 844-857, 1990.

[61] Motulsky H., *Intuitive Biostatistics*, New York, Oxford University Press, 1995.

[62] Munson J. and T. Khoshgoftaar, "The Detection of Fault-Prone Programs," *IEEE Transactions on Software Engineering*, vol. 18, no. 5, pp. 423-433, 1992.

[63] Musa J. D., *Software reliability engineering: More reliable software faster and cheaper Second Ed.*, Bloomington, Indiana, AuthorHouse, 2004.

[64] Nagappan N., L. Williams, and M. Vouk, "Towards a Metric Suite for Early Software Reliability Assessment," *International Symposium on Software Reliability Engineering Fast Abstract*, Denver, CO, 2003.

[65] Nagappan N., L. Williams, and M. Vouk, "Initial Results of Using In-Process Testing Metrics to Estimate Software Reliability," NCSU, Raleigh, NC, CSC TR-2004-5, January 25 2004.

[66] Nagappan N., L. Williams, M. Vouk, J. Hudepohl, and W. Snipes, "A Preliminary Investigation of Automated Software Inspection," *IEEE International Symposium on Software Reliability Engineering*, St. Malo, France, pp. 429-439, 2004.

[67] Nagappan N., L. Williams, M. Vouk, and J. Osborne, "Using In-Process Testing Metrics to Estimate Software Reliability: A Feasibility Study," *Fast Abstract at the International Symposium on Software Reliability Engineering*, St. Malo, France, pp. 21-22, November 2004.

[68] Nagappan N., "A Software Testing and Reliability Early Warning (STREW) Metric Suite," PhD Thesis in Computer Science at NCSU, Raleigh, NC 2005.

[69] Nagappan N. and T. Ball, "Use of Relative Code Churn Measures to Predict Defect Density," *ICSE*, St. Louis, MO, pp. 284-292, 15-21 May 2005.

[70] Nagappan N. and T. Ball, "Static Analysis Tools as Early Indicators of Pre-release Defect Density," *ICSE*, St. Louis, MO, pp. 580-586, 2005.

[71] Nagappan N., L. Williams, M. Osborne, M. Vouk, and P. Abrahamsson, "Providing Test Quality Feedback Using Static Source Code and Automatic Test Suite Metrics," *International Symposium on Software Reliability Engineering*, Chicago, IL, pp. 85-94, 2005.

[72] Nagappan N., T. Ball, and B. Murphy, "Using Historical In-Process and Product Metrics for Early Estimation of Software Failures," *ISSRE*, Raleigh, NC, pp. 62-74, May 2006.

[73] Nagappan N., T. Ball, and A. Zeller, "Mining metrics to predict component failures," *International Conference on Software Engineering*, Shanghai, China, May 20-28 2006.

[74] Neuhaus S., T. Zimmermann, C. Holler, and A. Zeller, "Predicting Vulnerable Software Components," *CCS*, Alexandria, VA, pp. 529-540, 29 October-2 November 2007.

[75] Offutt A. J., "The Coupling Effect: Fact or Fiction?," *International Symposium on Software Testing and Analysis*, Key West, Florida, pp. 131-140, 1989.

[76] Offutt A. J., "Investigations of the Software Testing Coupling Effect," *ACM Transactions on Software Engineering and Methodology*, vol. 1, no. 1, pp. 5-20, January 1992.

[77] Ostrand T. J., E. J. Weyuker, and R. M. Bell, "Where the bugs are," *ISSTA*, Boston, Massachusetts, pp. 86-96, 2004.

[78] Ozment A. and S. Schechter, "Milk or wine: does software security improve with age?," *15th Conference on USENIX Security Symposium*, pp. 93-104, July 2006.

[79] Prevelakis V. and D. Spinellis, "The Athens Affair," IEEE Spectrum, vol. 44, no. 7, pp. 26-33, July, 2007.

[80] Buffer Overflows – What Are They and What Can I Do About Them? http://www.cert.org/homeusers/buffer_overflow.html

[81] Rowe W. D., *An Anatomy of Risk*, Malabar, FL, Robert E. Krieger Publishing Co., 1988.

[82] Saltzer J. and M. Schroeder, "The Protection of Information in Computer Systems," *Proceedings of the IEEE*, vol. 63, no. 9, pp. 1278-1308, September 1975.

[83] SAS Institute Inc., "The Partition Platform," SAS Institute, Inc., Cary, NC, 2003.

[84] SAS Institute Inc., "Contingency Tables Analysis," SAS Institute, Inc., Cary, 2003.

[85] Schroter A., T. Zimmermann, and A. Zeller, "Predicting Component Failures at Design Time," *International Symposium on Empirical Software Engineering*, Rio de Janeiro, Brazil, pp. 18-27, September 21-22 2006.

[86] Shin Y. and L. Williams, "Is Complexity Really the Enemy of Software Security?," *Workshop on Quality of Protection*, Alexandria, VA, pp. 47-50, 2008.

[87] Stoneburner G., A. Goguen, and A. Feringa, "NIST Special Publication 800-30 - Risk Management Guide for Information Systems Technology," 2002.

[88] Subramanyan R. and M. S. Krisnan, "Empirical Analysis of CK Metrics for Object-Oriented Design Complexity: Implications for Software Defects," *IEEE Transactions on Software Engineering*, vol. 29, no. 4, pp. 297-310, April 2003.

[89] Tsipenyui K., B. Chess, and G. McGraw, "Seven Pernicious Kingdoms: A Taxonomy of Software Security Errors," *Automated Software Engineering*, Long Beach, CA, November 7-8 2005.

[90] Viega J. and G. McGraw, *Building Secure Software How to Avoid Security Problems the Right Way*, Boston, Addison-Wesley, 2002.

[91] Vouk M. and K. C. Tai, "Some Issues in Multi-Phase Software Reliability Modeling," *CASCON*, Toronto, pp. 512-523, October 1993.

[92] Walden J., A. Messer, and A. Kuhl, "Idea: Measuring the Effect of Code Complexity on Static Analysis Results," *Engineering Secure Software and Systems*, Leuven, Belgium, pp. 195-199, 4-6 February 2009.

[93] Williams L., M. Gegick, and A. Meneely, "Protection Poker: Structuring Software Security Risk Assessment and Knowledge Transfer," *ESSoS*, Leuven, Belgium, pp. 122-134, February 4-6 2009.

[94] Witten I. and E. Frank, *Data Mining*, Second ed. San Francisco, Elsevier, 2005.

[95] Wysopal C., L. Nelson, D. Dai Zovi, and E. Dustin, *The Art of Software Security Testing: Identifying Software Security Flaws*, Boston, Addison Wesley, 2006.

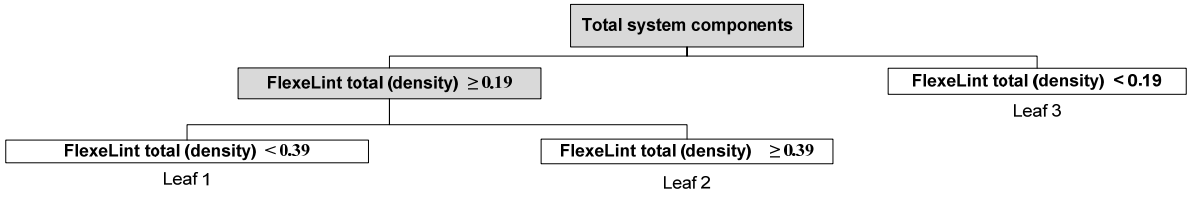
[96] Young M. and R. N. Taylor, "Rethinking the Taxonomy of Fault Detection Techniques," *ICSE*, pp. 53-62, 1989.

[97] Zheng J., L. Williams, W. Snipes, N. Nagappan, J. Hudepohl, and M. Vouk, "On the Value of Static Analysis Tools for Fault Detection," *IEEE Transactions on Software Engineering*, vol. 32, no. 4, pp. 240-253, April 2006.

Appendix

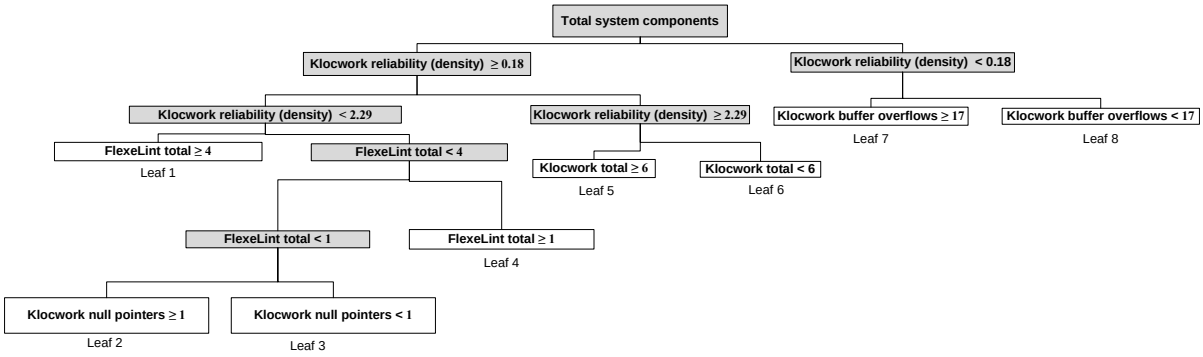
Appendix A

Case Study 1. CART model with SCSA warnings alone.



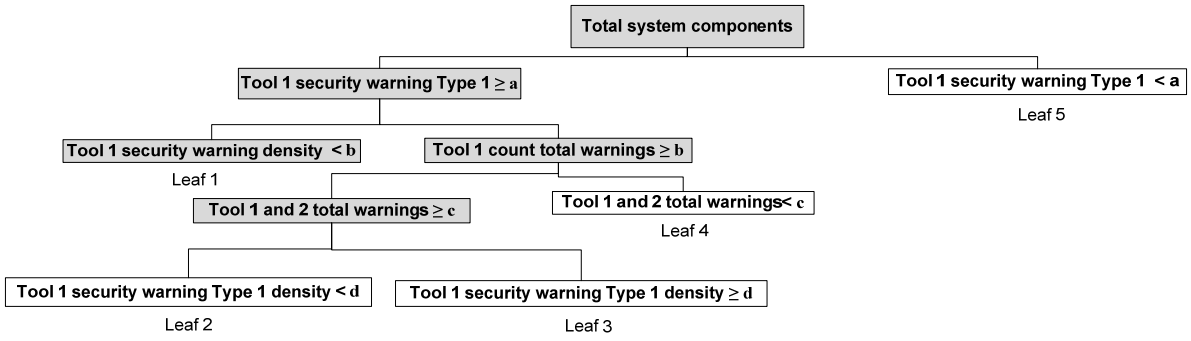
Appendix B

Case Study 2. CART model with SCSA warnings alone.



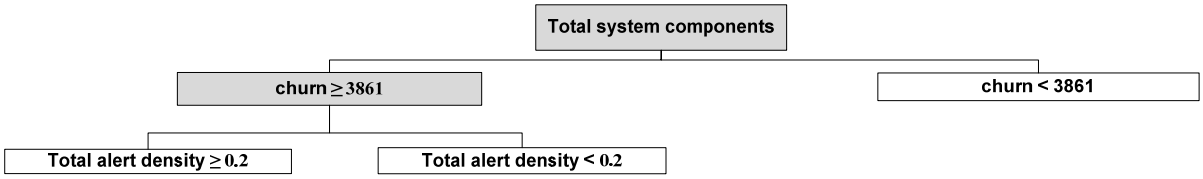
Appendix C

Case Study 3. CART model with SCSA warnings alone.



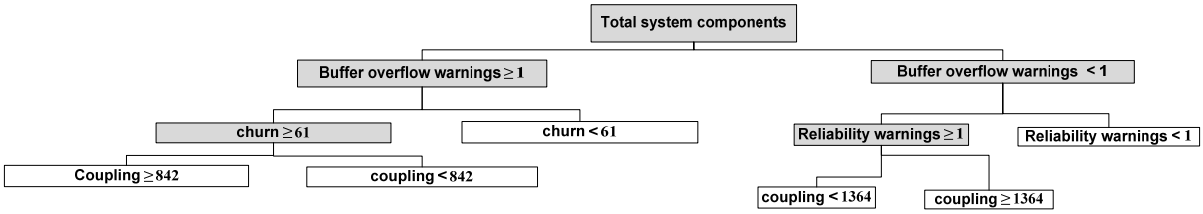
Appendix D

Case Study 1. CART model with SCSA warnings and churn.



Appendix E

Case Study 2. CART model with SCSA warnings, churn, coupling.



Appendix F

Case Study 3. CART model with SCSA warnings, churn, and faults found manually.